TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

HEIKKI KÄNÄ

SOVELLUSALUSTAN SUORITUSKYKYTESTAUKSEN AUTOMATISOINTI

Diplomityö

Tarkastaja: Prof. Hannu-Matti Järvinen
Tarkastaja ja aihe hyväksytty Tieto- ja sähkötekniikan tiedekuntaneuvoston kokouksessa 07.10.2015

TIIVISTELMÄ

HEIKKI KÄNÄ: Sovellusalustan suorituskykytestauksen automatisointi
Tampereen teknillinen yliopisto
Diplomityö, 66 sivua, 4 liitesivua
Huhtikuu 2016
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastaja: Prof. Hannu-Matti Järvinen
Avainsanat: suorituskykytestaus, testiautomaatio, sovellusalusta

Tietokoneohjelmat kehittyvät jatkuvasti monimutkaisemmaksi, minkä vuoksi niiden testauksen merkitys korostuu. Manuaaliseen ohjelmistojen testaukseen kuluu tavallisesti paljon aikaa ja resursseja. Testauksen kuluja ja testaukseen tarvittavaa työmäärää voidaan vähentää käyttämällä testiautomaatiota, jolloin osa testien suorituksesta jätetään automaattisen testaustyökalun tehtäväksi.

Suorituskykytestauksella selvitetään järjestelmän kykyä suoriutua tehtävästään kuorman alla, ja sen avulla voidaan havaita pullonkauloja, jotka heikentävät järjestelmän suorituskykyä. Järjestelmät, joiden suorituskyky ei ole riittävä, aiheuttavat ylimääräisiä kuluja ja vähentävät järjestelmästä saatavia hyötyjä. Tehokas ja luotettava suorituskykytestaus voidaan suorittaa vain automaattisen testaustyökalun avulla.

Sovellusalusta on joukko yhteisen rakenteen muodostavia alijärjestelmiä ja ohjelmointirajapintoja, joiden varaan sovelluksia voidaan kehittää. Tässä työssä toteutettiin eräälle sovellusalustalle suorituskykytestausjärjestelmä, joka mahdollistaa automaattisen suorituskykytestauksen. Työn tavoitteena oli selvittää, miten sovellusalustan suorituskykytestit ja niiden raportointi voidaan automatisoida, jotta saadaan hyödyllisiä ja vertailukelpoisia tuloksia järjestelmän suorituskyvystä.

Toteutettu suorituskykytestausjärjestelmä mahdollistaa sovellusalustan palvelinympäristön ja sovellusalustan varaan kehitettyjen sovellusten suorituskyvyn testaamisen. Testausjärjestelmä tarjoaa riittävät ominaisuudet suorituskykytestien toteuttamiseen ja tulosten tarkasteluun, mutta testausjärjestelmää voidaan edelleen kehittää monella eri tavalla. Testausjärjestelmällä on toteutettu kattavat suorituskykytestit sovellusalustan palvelinympäristölle, joten testausjärjestelmän toteutusta voidaan pitää onnistuneena.

ABSTRACT

HEIKKI KÄNÄ: Automating performance testing of an application platform
Tampere University of Technology
Master's Thesis, 66 pages, 4 appendix pages
April 2016
Master's Degree Programme in Information Technology
Major: Software engineering
Examiner: Prof. Hannu-Matti Järvinen
Keywords: performance testing, test automation, application platform

Computer programs are constantly becoming more complex and therefore the importance of testing is emphasized. Manual software testing usually takes a lot of time and resources. Testing costs and the amount of work required for testing can be reduced by using test automation. Using test automation a part of tests are left to be executed by an automated testing tool.

Performance testing can be used to figure out the system's ability to perform under load. It can also be used to detect bottlenecks, resulting in poor system performance. Systems which don't have sufficient performance can lead to extra costs and reduce the benefits gained from the system. Efficient and reliable performance testing can only be carried out by using an automated testing tool.

An application platform is a set of software subsystems and interfaces that form a common structure from which applications can be developed. In this thesis, a performance testing system was created for automating performance tests of an application platform. The aim was to find out how performance tests and result reporting of the application platform can be automated in order to provide useful and comparable results on the performance of the system.

The performance testing system developed in this thesis can be used to test performance of the application framework server. In addition, it can be used to test performance of applications developed for the platform. The testing system provides adequate means for creating and running performance tests but it can be further developed in different ways. A comprehensive set of performance tests has been created and executed for the application platform using the testing system developed, hence the implementation of the testing system can be considered a success.

ALKUSANAT

Olen tehnyt tämän diplomityön työskennellessäni Insta DefSec Oy:n palveluksessa. Haluan kiittää työni ohjaajaa Timo Kankaretta erinomaisista neuvoista ja ohjeista sekä työn tarkastajaa professori Hannu-Matti Järvistä työn huolellisesta tarkastamisesta. Lisäksi haluan kiittää kaikkia niitä henkilöitä, jotka ovat edesauttaneet työn valmistumista sekä koko projektiryhmää loistavan ja kannustavan työilmapiirin luomisesta.

Lopuksi haluan kiittää perhettäni, ystäviäni ja erityisesti Jennaa kaikesta avusta ja tuesta, jota olen saanut diplomityön kirjoittamisen ja opiskelujen aikana.

Tampereella 22. maaliskuuta 2016

Heikki Känä

SISÄLLYS

KUVALUETTELO

LYHENTEET JA MERKINNÄT

API	Application Programming Interface
FQDN	Fully Qualified Domain Name
HTML	HyperText Markup Language
JAR	Java Archive
JDK	Java Development Kit
JMX	Java Management Extensions
JPA	Java Persistence API
MBean	Managed Bean
OSGi	Open Services Gateway initiative
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SSH	Secure Shell
WSDL	Web Service Description Language
XLSX	Excel Microsoft Office Open XML Format Spreadsheet File
XML	Extensible Markup Language

1. JOHDANTO

Ohjelmistokehityksen alkuajoista lähtien tietokoneohjelmista on tullut koko ajan monimutkaisempia. Ohjelmistojen koon ja monimutkaisuuden lisääntyessä niiden testauksen merkitys korostuu. Perinteisesti testauksella tarkoitetaan suunnitelmallista virheiden etsimistä ohjelmasta, ja testauksen eri työvaiheisiin sekä virheiden jäljitykseen ja korjaamiseen kuluu tavallisesti yli puolet ohjelmistoprojektin resursseista [17, s. 205]. Ohjelmistojen testaus on kallista, mutta niin on myös liian vähäisen tai puutteellisen testauksen aiheuttamat virheelliset ohjelmat. Yhdysvaltain kauppaministeriön alaisen *National Institute of Standards and Technology* -viraston teettämän tutkimuksen mukaan puutteellisesta ohjelmistotestauksesta aiheutui Yhdysvalloissa vuonna 2000 59,5 miljardin dollarin kulut [25, s. ES-11].

Virheiden etsimisen lisäksi testauksella pyritään mittaamaan ohjelmiston laatua, kuten käytettävyyttä, luotettavuutta ja suorituskykyä. Suorituskykytestauksella selvitetään järjestelmän kykyä suoriutua tehtävästään kuorman alla. Riittävä suorituskyky on minkä tahansa järjestelmän kannalta tärkeää, ja järjestelmän systemaattisella suorituskykytestauksella voidaan havaita pullonkauloja, jotka heikentävät järjestelmän suorituskykyä. Järjestelmät, joiden suorituskyky ei ole riittävä, aiheuttavat ylimääräisiä kuluja ja vähentävät järjestelmästä saatavia hyötyjä [23].

Ohjelmistokehityksen tuottavuuden kasvu erilaisten teknologioiden ja työkalujen käytön myötä on lisännyt tarvetta testauksen automatisoinnille. Kun tuottavuus kasvaa, ohjelmista voidaan tehdä yhä monimutkaisempia. Monimutkaisempien ohjelmien myötä myös tarve testaukselle kasvaa, ja niiden testaus vaikeutuu. Testauksen tarpeen ja vaikeuden lisääntyminen tarkoittaa enemmän työtunteja, ja työtuntien lisääntyminen tarkoittaa enemmän kuluja. Manuaalinen ohjelmistojen testaus vaatii paljon aikaa, ja resursseiltaan rajatuissa ohjelmistoprojekteissa testaukseen tarvittavan työajan rajoittamiseksi voidaan käyttää automaattisia testaustyökaluja, joiden avulla testauksen vaatimaa työaikaa voidaan rajata huomattavasti.

Testauksen automatisoinnilla voidaan tehostaa ohjelmien testausta ja vähentää tes-

taajien työmäärää. Monimutkaisten järjestelmien tapauksessa testauksen automatisointi vaatii kuitenkin yleensä työkaluja, jotka on kehitetty erityisesti kyseistä järjestelmää varten. Testauksen automatisoinnin kehitykseen käytetyt resurssit voitetaan kuitenkin ajan kuluessa usein takaisin automatisoinnin tuoman tehokkuuden myötä.

Tässä diplomityössä esitellään sovellusalustaan kehitettävä suorituskykytestausjärjestelmä, jolla sovellusalustan palvelinympäristöä voidaan kuormittaa hallitusti suurella määrällä asiakassovelluksia. Testausjärjestelmällä mahdollistetaan automatisoitu suorituskykytestaus ja tulosten analysointi. Testausjärjestelmän avulla suoritetaan sovellusalustalle kattavat suorituskykytestit, ja lisäksi tarjotaan sovellusalustan varaan kehitetyille sovelluksille mahdollisuus testata sovellusten suorituskykyä. Tässä työssä selvitetään, miten sovellusalustan suorituskykytestit ja niiden raportointi voidaan automatisoida, jotta saadaan hyödyllisiä ja vertailukelpoisia tuloksia järjestelmän suorituskyvystä.

Diplomityö on jaettu kuuteen lukuun. Luvussa 2 tarkastellaan työn teoreettista taustaa ja selvitetään lukijalle testaukseen liittyviä käsitteitä ja termejä. Luvussa tarkastellaan testauksen automatisointia, suorituskykytestausta ja testaukseen käytettäviä työkaluja.

Luku 3 keskittyy toimintaympäristön kuvaukseen, jolle suorituskykytestipenkki toteutetaan. Luvussa kuvataan kohdejärjestelmänä toimiva sovellusalusta sekä kuvataan toteutettavan testausjärjestelmän tavoitteet ja vaatimukset.

Luvussa 4 esitellään toteutetun suorituskykytestausjärjestelmän rakenne. Luku kertoo yksityiskohtaisesti järjestelmän arkkitehtuurista ja suorituskykytestien määrittelystä ja suorituksesta.

Luku 5 keskittyy testausjärjestelmän käyttöön. Luvussa tarkastellaan, kuinka toteutetut suorituskykytestit suoritetaan, kuinka niiden etenemistä seurataan, ja minkälaisia tulosraportteja testausjärjestelmä tuottaa.

Luvussa 6 toteutettu testausjärjestelmä arvioidaan vertaamalla sitä järjestelmälle asetettuihin tavoitteisiin ja vaatimuksiin. Lisäksi luvussa kerrotaan testausjärjestelmän käyttökokemuksia sekä pohditaan testausjärjestelmän jatkokehitysmahdollisuuksia. Lopuksi luvussa 7 esitellään työn johtopäätökset.

2. OHJELMISTOJEN TESTAUS

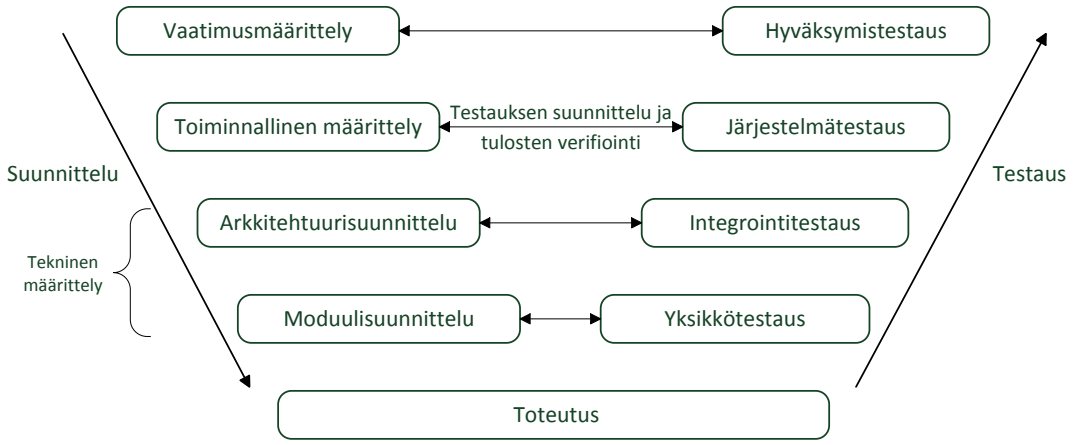
Tässä luvussa kerrotaan työn teoreettisesta taustasta ja selitetään aiheeseen liittyviä käsitteitä ja termejä. Aluksi ohjelmistojen testausta tarkastellaan yleisesti ja tehdään katsaus testauksen V-malliin. Seuraavaksi tarkastellaan testauksen automatisointia sekä suorituskykytestausta. Lopuksi luodaan katsaus erilaisiin testaustyökaluihin ja Java-ohjelmien testaukseen.

2.1 Testauksen merkitys

Ohjelmistojen testaus on empiirinen menetelmä, jossa testattavan kohteen laadusta hankitaan tietoa päätöksenteon tueksi. Testauksen tavoitteena on varmistaa, että ohjelmisto toimii riittävän hyvin sen kohdeympäristössä. Testaajat suorittavat testauksen yleensä manuaalisesti. *Manuaalisessa testauksessa* ohjelma testataan käsin, eikä apuna käytetä automaattisia työkaluja [6]. Manuaalinen testaus on yleisin testausmuoto, ja kaikesta testauksesta n. 75 % suoritetaan manuaalisesti [14].

Hyvän testausmenetelmän tulisi olla tehokas virheiden löytämisessä, ja samanaikaisesti testejä pitäisi pystyä suorittamaan riittävällä teholla ja riittävän halvalla [16, s. 3]. Kaikkea ei kuitenkaan voi testata. Mitä tahansa ei-triviaalia järjestelmää voidaan testata lukemattomilla määrillä eri syötteitä, ja testauksella niistä voidaan kattaa vain häviävän pieni osa [17, s. 205]. Esimerkiksi, mikäli ohjelma sisältää funktion, jolle tulee syöttää parametrina 16-bittinen kokonaisluku, mahdollisia syötteitä on 65536. Mikäli saman tyyppisiä parametreja on kaksi, on erilaisia syötevaihtoehtoja jo yli neljä miljardia, ja kyseessä on vain yksi funktio todennäköisesti lukemattomien muiden funktioiden joukossa. Testauksen avulla onkin mahdollista osoittaa, että ohjelmassa on virheitä, mutta ohjelman virheettömyyttä testauksella ei voida todistaa [17, s. 205].

Onnistunut testaus vaatii taitoa. Koska kaikkea ei voida testata, testien suunnittelu on hyvin tärkeä osa testausta. Koska kaikille järjestelmille on lukematon määrä eri-



Kuva 2.1 *Testauksen V-malli [17, 18]*

laisia testitapauksia, normaalin ohjelmistoprojektin puitteissa niistä voidaan testata hyvin pieni osa. Kuitenkin tämän pienen testijoukon oletetaan löytävän suurimman osan ohjelman virheistä. Tämän vuoksi oikean testin valinta on ensiarvoisen tärkeää, jotta testauksesta saadaan mahdollisimman korkea hyöty. [16, s. 4]

2.2 Testauksen V-malli

V-malli on yleisesti kehitysprojektien suunnittelu- ja toteutustöissä käytössä oleva prosessimallinnuskeino. Testauksessa V-mallia käytetään kehitystyön ja testauksen suhteen havainnollistamiseen, jonka mukaan kukin kehityksen vaihe vastaa tiettyä testauksen vaihetta [17, s. 206]. Lisäksi kunkin testausvaiheen tulokset todetaan oikeiksi vastaavalla suunnittelussa toteutetulla dokumentilla. Ohjelmistotuotannossa yleisesti käytetty V-malli on esitetty kuvassa 2.1.

V-mallin vasen puoli kuvaa ohjelmistotuotannon vesiputousmallia, jossa edetään vaiheittain ylhäältä alas vaatimusmäärittelystä toteutukseen. V-mallin oikealla puolella sijaitsevat kutakin vasemman puolen vaihetta vastaavat testauksen vaiheet. Vasenta puolta edetessä jokaisessa vaiheessa laaditaan vaihetta vastaava testaussuunnitelma. Kun ohjelman toteutus on edennyt siihen vaiheeseen, että sitä voidaan testata, edetään V-mallin oikeaa puolta alhaalta ylöspäin käyttäen aiemmin luotuja testaussuunnitelmia eri osien testien toteuttamiseen ja suorittamiseen. Testauksen jokaisessa vaiheessa varmistetaan, että toteutus vastaa määrittelyä. Esimerkiksi järjestelmätestauksella varmistetaan, että järjestelmän toteutus vastaa toiminnallista määrittelyä.

Vaikka V-malli yhdistetään usein vanhanaikaisena pidettyyn vesiputousmalliin, se tarjoaa edelleen hyvän abstraktion testauksen eri osien jaotteluun. V-mallia voidaan soveltaa myös nykyaikaisiin ketteriin menetelmiin, koska ketterien menetelmien iteraatiot voidaan ajatella lyhempinä versioina vesiputousmallin mukaisista projekteista [16, s. 6].

Löytyvien virheiden korjaaminen tulee sitä kalliimmaksi, mitä korkeammalla testaustasolla V-mallissa ollaan. Järjestelmätestauksessa havaitun virheen korjaaminen saattaa aiheuttaa muutoksia moneen järjestelmän osaan. Mikäli virhe olisi havaittu jo yksikkötasolla, olisi muutos pitänyt tehdä vain siihen komponenttiin, josta virhe löytyi. Tehty korjaus saattaa vaikuttaa ohjelman muihin osiin, tai aiheuttaa uusia virheitä, minkä vuoksi virheen korjaamisen jälkeen komponentin tai komponenttien testit tulee ajaa uudestaan. Tämän kaltaista ohjelman uudelleentestausta kutsutaan *regressiotestaukseksi*. [17, s. 208].

2.2.1 Yksikkötestaus

Yksikkötestauksessa testataan yksittäistä luokkaa tai moduulia, joka koostuu yleensä 100–1000 koodirivistä. Yksikkötestaus kohdistetaan ohjelman pienimpiin mahdollisiin testattaviin osiin, ja sen suorittaa yleensä testattavan yksikön toteuttaja. Usein yksikkötestaus on osa toteutustyötä, jolloin testaus suoritetaan kehitystyön yhteydessä [18].

Yksikkötestauksen tuloksia verrataan arkkitehtuuri- ja moduulisuunnittelun tuloksena syntyneisiin dokumentteihin, tavallisesti tekniseen määrittelyyn. Yksikkötestauksen toteuttamiseksi joudutaan usein toteuttamaan *testipetejä*, joilla yksikön toimivuutta voidaan kokeilla. Testipedit sisältävät ympäristöä simuloivia osia, jotka tarjoavat välttämättömät toiminnallisuudet yksikön testaukselle. Joissain tapauksissa testausta varten joudutaan toteuttamaan muiden luokkien toimintaa imitoivia *mock-olioita*. [17, s. 207]

2.2.2 Integrointitestaus

Integrointitestauksessa testataan yksittäisten yksiköiden muodostamien osakokonaisuuksien yhteistoimintaa ja rajapintoja. Tavallisesti integrointitestaus tapahtuu rinnakkain yksikkötestauksen kanssa. Integrointi voidaan suorittaa *kokoavasti* tai *jä-*

sentävästi. Kokoava integrointi etenee alimman tason yksiköistä ylöspäin. Jäsentävässä integroinnissa etenemissuunta on päinvastainen. Integrointitestauksen tuloksia verrataan tavallisesti tekniseen määrittelydokumenttiin. [17, s. 208]

Perinteisesti integrointi tehtiin esimerkiksi kerran viikossa, jolloin kehittäjien tuotokset yhdistettiin. Tässä niin kutsutussa *kertarysäysintegroinnissa* ilmenevät ongelmat on hankala paikantaa suuresta joukosta yksiköitä. Integrointien väli on sittemmin lyhentynyt, ja 2000-luvulla *jatkuva integrointi* alkoi yleistyä. Jatkuvassa integroinnissa pieni toteutettu osuus integroidaan suurempaan kokonaisuuteen heti sen valmistuttua. Näin mahdollisen vian ilmentyessä vika löydetään helposti, ja järjestelmä pysyy eheänä. [18]

2.2.3 Järjestelmätestaus

Järjestelmätestauksessa testauksen kohteena on koko järjestelmä. Järjestelmätestauksen tuloksia verrataan tavallisesti toiminnalliseen määrittelyyn ja muuhun asiakasdokumentaatioon. Järjestelmätestauksen yhteydessä suoritetaan yleensä toiminnallisten testien lisäksi ei-toiminnallisia ominaisuuksia testaavia testejä, kuten *suorituskyky-*, *asennus-* ja *käytettävyystestejä*. [17, s. 208]

Järjestelmätestaus on yleensä ajallisesti pitkäkestoisin testausvaihe, ja sen aikana löydettyjen virheiden korjaus on kallista. Ei-toiminnalliset testit mukaan luettuna koko järjestelmän testaus voi kuluttaa hyvin paljon resursseja. Kehittäjät suorittavat yleensä yksikkö- ja integrointitestauksen, mutta järjestelmätestaus on tavallisesti testaajien tai erillisen testausryhmän vastuulla. [18]

2.2.4 Hyväksymistestaus

Hyväksymistestauksella testataan, onko toteutettu ohjelmisto sopimusten mukainen, ja voidaanko se ottaa käyttöön. Hyväksymistestaus perustuu asiakkaan vaatimuksiin, ja sen suorittaa asiakas. Hyväksymistestaus suoritetaan ohjelmiston lopullisessa käyttöympäristössä. Hyväksymistestaus on yleensä lyhytkestoinen, koska varsinaisia virheitä ohjelmistosta ei enää etsitä. [18]

Hyväksymistestauksessa testataan valmista ohjelmistoa. Tavallisesti ohjelmiston loppukäyttäjät suorittavat hyväksymistestauksen. Hyväksymistestauksessa testataan

kaikkia ohjelmiston ominaisuuksia, kuten toiminnallisuutta, käytettävyyttä, suorituskykyä ja tietoturvaa. [18]

2.3 Testauksen automatisointi

Testauksen automatisoinnilla osa testien suorituksesta jätetään tietokoneen tehtäväksi, jolloin testien suorituksesta huolehtii automaattinen testausohjelma. Automatisoimalla testien suoritus testaukseen käytettävää aikaa saadaan vähennettyä tai vastaavasti samassa ajassa ajettavien testien määrää voidaan kasvattaa. Automaattisilla testeillä saadaan luotettavia, toistettavia tuloksia, koska testeissä käytetään aina täsmälleen samoja syötteitä, ja ihmisen aiheuttama inhimillinen virhe saadaan poistettua testien suorituksesta. Automaattisen testauksen myötä saavutettava säästö voi olla jopa 80 % testaukseen käytettävistä kuluista verrattuna manuaaliseen testaukseen. Aina säästöä ei kuitenkaan synny, mutta testien automatisoinnilla ohjelmistojen laatua voidaan parantaa. [16, s. 3]

Testauksen automaation käyttökohteita ovat yleisesti sellaiset testit, jotka on mahdollista suorittaa toistuvasti samalla tavalla. Toistuvat, yksinkertaiset, ihmiselle tylsät testit ovat hyviä automatisoinnin kohteita, koska tietokone suorittaa ne aina samalla tavalla. Tällaisia testejä ovat muun muassa regressiotestit ja suorituskykytestit. [14]

2.3.1 Hyödyt

Automaattisella testauksella saavutetaan monia hyötyjä manuaaliseen testaukseen verrattuna. Ilmeinen hyöty testiautomaation käytöstä saavutetaan regressiotestauksen yhteydessä. Kun testattavaan järjestelmään tulee muutos, automaattisilla testeillä pientenkin muutosten vaikutus ohjelmistoon voidaan testata vaivattomasti. [16, s. 9]

Automaattiset regressiotestit luovat kehittäjille turvaverkon, joka varmistaa, että ohjelmaan tehdyt muutokset eivät riko ohjelman muuta toiminnallisuutta. Testit antavat myös välittömän palautteen: mikäli muutos ohjelmassa aiheuttaa sen, että jokin testi ei mene läpi, korjaus tehtyyn ohjelmakoodiin voidaan tehdä nopeasti. Automaattiset testit toimivat myös ohjelman dokumentaationa, sillä ne kuvaavat ohjelman odotettua toimintaa. Automaattisen testin epäonnistuminen viestii kehittäjälle, että ohjelma ei toimi oikein. [15, ss. 257–264]

Manuaaliseen testaukseen verrattuna automaation avulla testejä saadaan ajettua enemmän lyhemmässä ajassa [26]. Lisäksi testiautomaation avulla voidaan toteuttaa testejä, joiden suorittaminen olisi mahdotonta manuaalisella testauksella. Esimerkiksi järjestelmän testaaminen 200 yhtäaikaisella käyttäjällä voi olla manuaalisella testauksella mahdotonta, mutta testiautomaatiota käyttäen tilannetta voidaan simuloida ja saada vastaavia tuloksia kuin todellisessa käytössä [16, s. 9].

Automaatio tehostaa testaukseen käytettävien resurssien käyttöä ja vähentää testauksen vaatimia kustannuksia. Testaajien työaikaa ei tarvitse käyttää samojen, toistuvien testien ajoon manuaalisesti, vaan testiautomaatiojärjestelmä huolehtii niiden suorittamisesta. Näin testaajat voivat käyttää työaikansa esimerkiksi uusien testien suunnitteluun. Lisäksi tietokoneita voidaan käyttää testien suorittamiseen työajan ulkopuolella, esimerkiksi öisin ja viikonloppuisin. [26]

Testiautomaatio parantaa testien yhtenäisyyttä ja toistettavuutta. Koska testien ajo jätetään tietokoneen tehtäväksi, katoaa testausprosessista ihmisen aiheuttama inhimillinen virhe. Testit ajetaan joka kerta täsmälleen samoilla syötteillä, jolloin kullekin testaajalle ominaiset tavat eivät vaikuta testituloksiin [16, s. 9]. Samoja automatisoituja testejä voidaan myös ajaa erilaisilla järjestelmäkonfiguraatioilla, esimerkiksi eri tietokannalla tai käyttöjärjestelmällä. Näin monissa eri laiteympäristöissä käytettävien järjestelmien testaus tehostuu, ja eri konfiguraatioilla ajetuista testeistä saadaan yhtenäisiä tuloksia [26].

2.3.2 Ongelmat

Testausautomaation on havaittu hyödyntävän useita testauksen osa-alueita. Siihen liittyy kuitenkin myös joitain ongelmia. Testausautomaation käyttöönotto vaatii usein suuren alkuinvestoinnin. Tämä liittyy erityisesti testauksen kohteena oleviin monimutkaisiin, mukautettuihin järjestelmiin, joissa ei voida hyödyntää valmiita testiautomaatiojärjestelmiä [26]. Testaustyökalujen vertailu ja valinta tai mahdollisen oman testaustyökalun toteutus vaativat resursseja. Ilman tarkkaa taustatyötä valitun työkalun ominaisuudet saattavat osoittautua puutteelliseksi, jolloin siihen mahdollisesti käytetyt resurssit ovat menneet hukkaan [15, ss. 264–270].

Tehokas testiautomaation hyödyntäminen vaatii paljon opettelua. Testiautomaation käyttöönottoon liittyy usein paljon uuden oppimista, mikäli testaajat eivät ole hyödyntäneet automaatiota ennen. Testien automatisointiin voi liittyä myös liiallista

turvallisuuden tunnetta. Testien läpäisy ei tarkoita, että ohjelma olisi virheetön. Testitapaukset saattavat olla huonosti suunniteltuja, eivätkä ne välttämättä testaa oikeita asioita. [16, s. 10-11]

Automaattisia testejä tulee myös pitää yllä. Usein järjestelmän toiminnallisuuden muuttuessa joitain tai kaikkia automaattisia testejä pitää muuttaa vastaavasti, jotta muuttunut järjestelmä läpäisee testit. Pienikin ohjelmakoodiin tehty muutos saattaa aiheuttaa usean automaattisen testin muuttamista, jolloin testien ylläpitäminen voi huonossa tapauksessa viedä suurimman osan muutokseen vaadittavasta ajasta. [16, s. 10-11]

Automaation käyttöönoton myötä löytyvien virheiden määrälle voi myös olla liian suuria odotuksia. Testiautomaatiolla toistetaan samoja testitapauksia, ja samojen testien helppo toistaminen on erityisesti regressiotestauksessa hyödyllistä. Testitapaus löytää virheen todennäköisimmin ensimmäisellä testiajolla, ja mikäli testiä ajetaan samassa, muuttumattomassa ympäristössä useita kertoja, virheen löytymisen mahdollisuus on ensimmäisen ajon jälkeen pieni. [16, s. 10-11]

2.4 Suorituskykytestaus

Suorituskykytestaus on ohjelmistojen testauksen osa, jolla mitataan testattavan järjestelmän ominaisuuksia kuorman alla. Suorituskykytestausta käytetään muun muassa testattavan kohteen valmiuden arviointiin, asetettujen suorituskykyvaatimusten täyttymisen arviointiin sekä erilaisten järjestelmäkonfiguraatioiden tehokkuuden arviointiin. Lisäksi järjestelmän systemaattisella suorituskykytestauksella voidaan kartoittaa järjestelmän laitteistokapasiteetin tarvetta sekä havaita pullonkauloja järjestelmän suorituskyvyssä. [22]

Suorituskykytestaus suoritetaan järjestelmä- tai hyväksymistestauksen yhteydessä. Suorituskykytestauksessa testausympäristön tulisi vastata mahdollisimman tarkasti todellista käyttöympäristöä. Realististen tulosten saamiseksi suorituskykytestaus tulee suorittaa koko järjestelmälle. Yksittäisten komponenttien suorituskykytestaus on myös hyödyllistä, mutta mahdollisten komponenttien yhteistoiminnasta aiheutuvien vaikutusten vuoksi niistä saatavat mittaustulokset saattavat antaa liian optimistisen kuvan koko järjestelmän suorituskyvystä. [23]

Ohjelmistojen suorituskykytestaus on edelleen hyvin vapaamuotoista ja epävirallista, toisin kuin muussa testauksessa, jossa alalle on kehittynyt joukko vakiintuneita

menetelmiä. Usein järjestelmän suorituskykytestaus jätetään projektin loppuun, eikä siihen käytetä riittävästi resursseja. Tästä voi seurata se, että järjestelmän suorituskyvyn pullonkauloja ei havaita, tai niitä ei ehditä korjata tuotteen toimitukseen mennessä. Suorituskykytestauksen alkaessa järjestelmän tulee olla riittävän vakaa. Mikäli järjestelmän toiminta on puutteellista tai virheellistä, ei suorituskykytestauksella saada luotettavia tuloksia. Tehokasta yksikkö- ja toiminnallista testausta voidaan pitää onnistuneen suorituskykytestauksen esiehtona. [23, s. 9]

Tehokas ja luotettava suorituskykytestaus voidaan suorittaa vain automaattisen työkalun avulla. Esimerkiksi useiden rinnakkaisten käyttäjien aiheuttamaa kuormaa on lähes mahdotonta testata kohtuullisella vaivalla ilman automaatiota. Lisäksi ihmisten aiheuttamien virheiden vuoksi tällaisen testin tulokset eivät olisi toistettavia. Suorituskykytestauksessa käytettävien työkalujen avulla voidaan usein nauhoittaa käyttäjän syötteitä, joiden perusteella luodaan järjestelmän normaalia käyttöä vastaavia testejä. Ominaista suorituskykytestaustyökaluille on lisäksi tulosten analysoinnin automatisointi, jonka avulla tuloksista luodaan automaattisia raportteja. Näitä raportteja voidaan helposti verrata aikaisemmin ajettujen testien raportteihin. [23, s. 12]

2.4.1 Suorituskyky

Suorituskyky on subjektiivinen, loppukäyttäjän näkökulmasta havaittu ominaisuus, joka vaihtelee eri käyttäjien välillä. Yhden käyttäjän mielestä suorituskykyinen ohjelma voi toisen käyttäjän mielestä olla käyttökelvoton. Suorituskykyinen ohjelma on sellainen, joka mahdollistaa haluttujen toimintojen suorittamisen ilman aiheetonta havaittua viivettä tai ärsyyntymistä [23, s. 1]. Suorituskykyisen ohjelman käytön tulee siis olla sellaisella tavalla viiveetöntä, että käyttäjän huomio ei herpaannu suoritettavasta toiminnosta.

Hyvälle tai huonolle suorituskyvylle ei ole alan hyväksymää virallista määritelmää [23, s. 3]. Riittävä suorituskyky riippuu hyvin paljon suoritettavasta tehtävästä. Esimerkiksi, jos tilauksen lisääminen järjestelmään aiheuttaa kahden sekunnin viiveen, järjestelmää voidaan usein pitää riittävän suorituskykyisenä. Toisaalta, jos tekstinkäsittelyohjelmassa syötetyn merkin ilmestyminen ruudulle kestää kaksi sekuntia, on ohjelma todennäköisesti suurimmalle osalle käyttäjistä käyttökelvoton.

2.4.2 Tarkoitus ja hyödyt

Suorituskykytestauksella voidaan arvioida seuraavia asioita testattavasta järjestelmästä [22]:

- järjestelmän valmiusaste
- infrastruktuurin riittävyys
- muutosten vaikutus suorituskykyyn
- järjestelmän konfiguraatioiden vaikutus.

Suorituskykytestauksella voidaan arvioida testattavan järjestelmän valmiutta tuotantokäyttöönoton kannalta. Kattavien suorituskykytestien perusteella voidaan päätellä, onko järjestelmän suorituskyky riittävä julkaisua varten ja riittääkö suorituskyky myös tulevaisuuden tarpeisiin. Lisäksi suorituskykytestien tuloksista voidaan arvioida järjestelmän saatavuusastetta ja vasteaikoja. [22]

Suorituskykytestauksella voidaan myös hankkia tietoa järjestelmän infrastruktuurin riittävyydestä. Testauksella voidaan arvioida järjestelmän kapasiteetin raja-arvoja sekä tarvittaessa kartoittaa järjestelmän vaatimien resurssien määrää, joilla riittävä suorituskyky saavutetaan. Suorituskykytestausta voidaan lisäksi käyttää erilaisten järjestelmäkonfiguraatioiden suorituskykyvaikutusten testaamiseen suorittamalla samat testit eri konfiguraatioille. [22]

Järjestelmään tehtävien muutosten vaikutusta suorituskykyyn voidaan myös testata suorittamalla järjestelmälle suorituskykytestit ennen muutoksen käyttöönottoa ja sen jälkeen. Näin tehdyn muutoksen suorituskykyvaikutuksesta saadaan nopea palaute, ja muutoksen vaikutusta suorituskykyyn voidaan verrata aikaisempiin tuloksiin. Suorituskykytestauksen avulla voidaan myös tehostaa järjestelmän konfiguraatioparametrien vaikutusta testaamalla niiden vaikutusta suorituskykyyn ennen tuotantokäyttöönottoa. [22]

2.4.3 Suorituskyvyn mittaaminen

Suorituskyvyn mittaamiseen voidaan käyttää erilaisia suureita, jotka voidaan jakaa kahteen osaan. Järjestelmän palvelukykyisyyteen liittyvät *saatavuus* (engl. availabi-

lity) ja *vasteaika* (engl. response time) mittaavat järjestelmän suorituskykyä käyttäjänäkökulmasta. Tehokkuuteen liittyviä suureita ovat *suoritusteho* (engl. throughput) ja *käyttöaste* (engl. utilization), joilla mitataan, kuinka järjestelmä käyttää toimintaympäristöään. [23, s. 2]

Suureet voidaan määritellä seuraavasti [23, s. 3, ss. 37–43]:

- **Saatavuudella** tarkoitetaan aikaa, jonka järjestelmä on käytettävissä loppukäyttäjänäkökulmasta. Järjestelmä ei ole saatavilla, mikäli käyttäjä ei pysty käyttämään sitä johtuen joko liian pitkistä viiveistä tai sen saavuttamattomuudesta.

- **Vasteaika** on syötteestä vastauksen saapumiseen kuluva aika. Suorituskykytestauksessa mitataan useimmiten vasteaikaa käyttäjän näkökulmasta, eli kestoa käyttäjän tekemästä syötteestä järjestelmän lähettämään vastaukseen.

- **Suoritusteholla** mitataan, kuinka monta toimintoa järjestelmä kykenee suorittamaan annetussa ajassa.

- **Käyttöasteella** tarkoitetaan sitä osuutta resursseista, joka mittauksen kohteena olevalla järjestelmällä on käytössään. Yleisimmät mitattavat asiat ovat suorittimen, keskusmuistin ja siirräntälaitteiden käyttöasteet.

2.4.4 Suorituskykytestauksen eri muodot

Suorituskykytestauksen, kuten toiminnallisenkin testauksen, tapauksessa hyödyllisten testitapausten suunnittelu on tärkeää. Sovelluksia voidaan testata suorituskykynäkökulmasta lukemattomilla eri tavoilla. Alalle on vakiintunut joukko termejä, joilla kuvataan suorituskykytestauksen yleisimpiä muotoja. Suorituskykytestauksen yleisimmät muodot ovat *kuormitustestaus* (engl. *load / volume testing*), *rasitustestaus* (engl. *stress testing*) ja *kestävyystestaus* (engl. *soak / stability testing*).

Kuormitustestaus on suorituskykytestauksen yleisin muoto, ja sillä testataan järjestelmän toimintaa odotetulla kuormalla. Kuormitustestauksen tavoitteena on saada tietoa testattavan järjestelmän saatavuudesta, suoritustehosta ja vasteajasta. Kuormitustestauksella järjestelmää päästään yleensä testaamaan lähellä todellista käyttöympäristöä ja usein sen yhteydessä testataan myös käyttäjälle näkyviä vaikutuk-

sia, kuten viiveitä tai käyttökatkoja. [23, s. 50]

Rasitustestauksella järjestelmää testataan odotettua suuremmalla kuormalla. Rasitustestauksen tavoitteena on aiheuttaa järjestelmälle niin suuri kuorma, että sitä ei voi enää käyttää. Näin saadaan tietoa järjestelmän kapasiteetin ylärajasta, eli esimerkiksi kuinka suurella käyttäjämäärällä järjestelmää voidaan vielä käyttää, ennen kuin järjestelmä muuttuu käyttökelvottomaksi. Rasitustestauksessa kuormaa lisätään niin pitkään, että jokin osa järjestelmästä ei toimi enää riittävän tehokkaasti, esimerkiksi vasteaika kasvaa liian suureksi, tai käyttäjät eivät saa enää yhteyttä järjestelmään. Rasitustestauksella mitataan suorituskyvyn lisäksi järjestelmän kapasiteettia. [23, s. 50]

Kestävyystestauksella järjestelmän toimintaa testataan pitkiä aikoja jatkuvalla kuormalla. Kestävyystestauksen tavoitteena on havaita järjestelmän suorituskyvyssä ilmeneviä ongelmia, jotka tulevat esille vasta pitkällä aikavälillä. Tällaisia ongelmia voivat olla esimerkiksi muistivuodon aiheuttama muistin loppuminen tai järjestelmän osien välisten yhteyksien auki jäämisestä johtuva järjestelmän jumiutuminen. Järjestelmän vasteajat saattavat myös pidentyä ajan myötä. [23, s. 51] Tällaisten ongelmien havaitsemiseksi kestävyystestaus tulee suorittaa ympäristössä, jonka käyttäytymisestä ja resursseista saadaan riittävästi tietoa. Esimerkiksi muistivuotojen havaitsemiseksi järjestelmässä tulee usein olla erillinen monitorointiohjelmisto, joka havaitsee resurssien, kuten suorittimen tai muistin, käyttöasteen.

Suorituskykytesteillä järjestelmän toimintaa testataan usein tuottamalla suuri määrä rinnakkaisia asiakasyhteyksiä testattavaan järjestelmään. Asiakasyhteydet voidaan luoda samanaikaisesti, jolloin saadaan tietoa järjestelmän toiminnasta tietyllä asiakasmäärällä. Asiakasyhteyksien määrää voidaan myös kasvattaa tasaisesti, jolloin puhutaan ns. *ramp-up*-testistä. Jatkuvasti muuttuvan kuorman vuoksi ramp-up-testeistä ei saada toistettavia suorituskykytuloksia, mutta niiden avulla saadaan nopeasti tietoa järjestelmän käyttäytymisestä eri kuormilla. Tätä tietoa voidaan myöhemmin hyödyntää kuormitus- ja rasitustestien kuormien määrittämisessä. [20]

2.5 Testaustyökalut

Testauksen automatisointi ja suorituskykytestaus perustuvat ohjelmallisten testaustyökalujen käyttöön. Erilaisia testaustyökaluja on tarjolla V-mallin mukaisen testauksen jokaiseen vaiheeseen.

Testauksen suunnittelun yhteydessä voidaan käyttää *testitapausgeneraattoreita*, joilla testejä voidaan tuottaa suoraan ohjelman spesifikaation tai rajapintojen avulla [16]. Usein testitapausten generointi on kuitenkin mahdotonta, koska se vaatii testattavan ohjelman formaalia määrittelyä [17].

Toteutuksen ja yksikkötestauksen yhteydessä voidaan käyttää *staattisia koodianalysointityökaluja* sekä *testikattavuustyökaluja*. Koodin analysointityökalut analysoivat koodia ilman sen suorittamista ja niiden avulla koodista voidaan mitata erilaisia suureita ja havaita mahdollisia virheitä [16]. Testikattavuustyökalulla mitataan, kuinka suuri osa testattavasta ohjelmakoodista on suoritettu esimerkiksi yksikkötestien yhteydessä. Testikattavuustyökaluilla voidaan myös mitata koodirivien suorituskertojen lisäksi prosessoriajan käyttöä [17].

Integrointitestauksen yhteydessä voidaan hyödyntää *dynaamisia analysointityökaluja*. Dynaamisilla analysointityökaluilla voidaan analysoida järjestelmää ajonaikaisesti. Näillä työkaluilla voidaan havaita esimerkiksi muistivuotoja. [16]

Järjestelmä- ja hyväksymistestauksessa voidaan käyttää *simulaattori-* ja *suorituskykytestaustyökaluja*. Simulaattorit mahdollistavat sellaisten tilanteiden testaamisen, joiden testaaminen ei ole reaalimaailmassa mahdollista. Suorituskykytestaukseen tarkoitetut työkalut soveltuvat hallitun kuorman luomiseen, ja niillä voidaan mitata esimerkiksi järjestelmän vasteaikaa ja käyttöastetta. Näillä työkaluilla voidaan usein myös testata järjestelmän toimintaa monella yhtäaikaisella käyttäjällä luomalla rinnakkaista kuormaa testattavaan järjestelmään. [16]

Aina markkinoilta löytyvät testaustyökalut eivät sovellu käytettäväksi järjestelmässä esimerkiksi riittämättömien ominaisuuksien vuoksi. Tällaisissa tilanteissa paras ratkaisu voi olla oman testaustyökalun kehittäminen. Oman testaustyökalun kehittäminen vaatii usein suuren alkuinvestoinnin, mutta sillä saavutetaan myös hyötyjä. Itse toteutettu työkalu sisältää juuri ne halutut ominaisuudet, joita järjestelmä vaatii. Lisäksi työkalu voidaan kehittää erityisesti tiettyä järjestelmää ajatellen, jolloin sen uudelleenkäyttömahdollisuudet mahdollisesti vähenevät, mutta testien toteutus nopeutuu. [16]

2.6 Java-ohjelmien testaus

Java-ohjelmien testaukseen on saatavilla erilaisia työkaluja. Suosittuja Java-ohjelmien yksikkötestaustyökaluja ovat muun muassa JUnit ([3]) ja TestNG ([11]).

Taulukko 2.1 *JUnitin yleisimmät annotaatiot*

Metodin annotaatio	**Selitys**
`@Test`	Testimetodi
`@Before`	Suoritetaan ennen jokaista testimetodia
`@After`	Suoritetaan jokaisen testimetodin jälkeen
`@BeforeClass`	Suoritetaan ennen ensimmäistä testimetodia
`@AfterClass`	Suoritetaan viimeisen testimetodin jälkeen

JUnit on xUnit-yksikkötestaustuoteperheeseen kuuluva Javalle kehitetty yksikkötestauskehys [3]. JUnit on suosituin Javan testaustyökaluista, ja vuonna 2013 tehdyn tutkimuksen mukaan 30000 GitHub-projektista 30,7 % käytti JUnitia [27]. JUnit-testit määritellään Java-luokkina, jotka sisältävät JUnit-annotoituja metodeita. Annotoidut metodit suoritetaan testien ajamisen yhteydessä taulukon 2.1 mukaisesti.

Javalle on tarjolla myös suorituskykytestaukseen liittyviä työkaluja. Apache JMeter on suosittu Java-ohjelmien kuormitustestaukseen kehitetty työkalu, jolla voidaan testata muun muassa web-palvelujen suorituskykyä [2]. JMeter mahdollistaa myös visuaalisten analyysien tekemisen testattavan kohteen suorituskyvystä. Lisäksi Javan JDK (*Java Development Kit*)-kehitysversion mukana tulevalla JConsole-työkalulla voidaan monitoroida Java-ohjelmien suoritukseen liittyviä suureita, kuten säikeiden määrää ja muistin käyttöä [13].

3. TOIMINTAYMPÄRISTÖN JA TOTEUTETTAVAN JÄRJESTELMÄN KUVAUS

Tässä luvussa esitellään tässä työssä kuvatun suorituskykytestausjärjestelmän toimintaympäristö ja kuvataan toteutettava järjestelmä. Aluksi luvussa tehdään katsaus sovellusalustoihin, minkä jälkeen kuvataan kohdejärjestelmä, johon suorituskykytestausjärjestelmä toteutetaan. Lopuksi luvussa kuvataan toteutettavan testausjärjestelmän tavoitteet ja vaatimukset sekä kartoitetaan mahdollisten valmiiden testaustyökalujen käyttöä.

3.1 Sovellusalustat

Sovellusalusta on joukko yhteisen rakenteen muodostavia alijärjestelmiä ja ohjelmointirajapintoja, joiden varaan sovelluksia voidaan kehittää. *Alustalla* tarkoitetaan tietotekniikassa yleisesti tietokoneohjelmien ajoympäristön muodostavaa järjestelmää. Tällä alustalla voidaan tarkoittaa kuvauksen tasosta riippuen tietokoneen suoritinta tai laitteiston ja käyttöjärjestelmän muodostavaa kokonaisuutta. Ohjelmistoarkkitehtuurien yhteydessä sanan alusta merkitys poikkeaa jossain määrin yleisesti käytetystä. Tietokoneohjelman arkkitehtuuri koostuu usein kerroksista. Yhden kerroksen näkökulmasta sen alla olevat kerrokset muodostavat kyseisen kerroksen alustan. Alustan tarjoamat toiminnot riippuvat kerroksen abstraktiotasosta tietokonejärjestelmässä. [24, s. 15].

Sovellusalustat tarjoavat siis niiden varaan kehitettäville sovelluksille alempien kerrosten mukanaan tuoman infrastruktuurin. Lisäksi ne tarjoavat sovellusten kehittämistä helpottavia rajapintoja. Sovellusalustojen avulla sovellusten kehittäminen nopeutuu, sillä alustan tarjoamien perustoiminnallisuuksien myötä sovellukset vaativat vähemmän ohjelmakoodia ja testausta ja niiden kehittäminen on edullisempaa [12].

3.2 Kohdejärjestelmä

Järjestelmä, johon testausjärjestelmä kehitetään, on Java-pohjainen sovellus- ja teknologia-alusta (myöhemmin sovellusalusta). Sen on tarkoitus toimia yleiskäyttöisenä johtamisjärjestelmäalustana. Järjestelmä ei tule sellaisenaan käyttöön, vaan sen varaan rakennetaan varsinaiset toimialakohtaiset loppukäyttäjäsovellukset.

Sovellusalusta tarjoaa loppukäyttäjäsovelluksille teknologiapinon, alustan perustoiminnallisuuden ja joukon alustan varaan rakennettuja yleiskäyttöisiä palveluita. Teknologiapino koostuu joukosta eri tarkoituksiin soveltuvia yleisiä kirjastoja, joita sovellukset voivat hyödyntää. Alustan perustoiminnallisuudet kattavat muun muassa tiedon pysyvään tallentamiseen, hajautukseen, sovellusten konfigurointiin, järjestelmänhallintaan ja tietoturvaan liittyvät palvelut.

Sovellusalustaa käytetään erilaisissa laiteympäristöissä, jolloin sen tulee tarjota riittävä suorituskyky ympäristöissä, joissa laitekapasiteetti vaihtelee. Järjestelmän käyttöympäristönä voi toimia konesali, jolloin järjestelmä tarjoaa suorituskykyisen ajoympäristön sovelluksille. Kevyimpänä laiteympäristönä sovellusalustalle toimii yksittäinen kannettava tietokone, jolloin järjestelmän tulee olla myös käytettävissä kohtuullisella sovelluskuormalla.

3.2.1 Arkkitehtuuri

Järjestelmä perustuu asiakas-palvelin-arkkitehtuuriin, ja se on karkeasti myös palvelupohjaisen SOA (*Service Oriented Architecture*) -arkkitehtuurin mukainen. Järjestelmä koostuu palvelin- ja asiakasympäristöistä. Loppukäyttäjän sovelluksille voidaan toteuttaa sekä palvelin- että asiakaskomponentti, ja ne voivat kommunikoida keskenään palvelupohjaisen arkkitehtuurin mukaisesti niin, että palvelin julkaisee palveluja, joita asiakas voi käyttää. Myös itse sovellusalustan tarjoamat palvelut julkaistaan palveluina, joita voidaan kutsua joko palvelinympäristöstä tai etäkutsuja käyttämällä asiakasympäristöstä.

Järjestelmä on toteutettu pääasiassa Java-ohjelmointikielellä. Javan nykyinen versio ei kuitenkaan tarjoa monimutkaisten järjestelmien kehittämistä helpottavaa moduulijärjestelmää. Kieleen on kuitenkin tarjolla OSGi Alliance -organisaation kehittämä OSGi-standardi (*Open Services Gateway initiative*), joka tarjoaa Javaan moduulijärjestelmän. OSGi on tällä hetkellä käytetyin Javan moduulijärjestelmä, ja

standardilla on useita toteutuksia [7]. OSGin toiminta perustuu OSGi-paketteihin (engl. *bundle*), jotka ovat normaaleja Javan JAR-tiedostoja (*Java Archive*), joissa on lisäksi erityinen manifestitiedosto moduulin määrittämiseksi. Tässä tiedostossa määritellään moduulin muille OSGi-paketeille tarjoamat palvelut sekä riippuvuudet. Muille paketeille näkyvät siis vain manifestissa määritellyt palvelut, jolloin palveluiden toteutusluokkia saadaan kapseloitua OSGin mekanismein. Sovellusalustan ajoympäristö perustuu OSGi-moduulijärjestelmään. Suurin osa asiakas- ja palvelinympäristön Java-luokista on pakattu OSGi-paketeiksi.

3.2.2 Palvelut

Sovellusalusta tarjoaa loppukäyttäjäsovellusten toteuttamiseen erilaisia palveluita ja teknologioita. Palvelut sisältävät sovellusten kehittämistä helpottavia toiminnallisuuksia. Työssä toteutettava testausjärjestelmä hyödyntää toteutuksessaan joitain sovellusalustan tarjoamista palveluista. Näitä palveluita tarkastellaan seuraavaksi. Muiden sovellusalustan tarjoamien palveluiden esittely jää tämän työn ulkopuolelle.

Asiakas- ja palvelinympäristön sovellukset ja palvelut kommunikoivat sovellusalustan interaktiokehyksen avulla. Interaktiokehys koostuu kahdesta osasta: etäkutsukehyksestä ja notifikaatiopalvelusta.

Etäkutsukehys
Etäkutsukehyksen avulla asiakassovellukset voivat kutsua verkon välityksellä palvelinympäristön julkaisemia palveluita. Etäkutsut on toteutettu käyttäen SOAP-protokollaa (*Simple Object Access Protocol*) ja WSDL-palvelunkuvauskieltä (*Web Service Description Language*). Etäkutsukehyksen avulla palvelinympäristön palveluiden ohjelmointirajapinnat saadaan asiakassovellusten käyttöön siten, että niiden kutsuminen tapahtuu paikallisten rajapintojen tapaan.

Notifikaatiopalvelu
Notifikaatiopalvelu mahdollistaa palvelinympäristössä tapahtuvien muutosten julkaisun asiakkaille sekä muille palvelinympäristön palveluille. Notifikaatiopalvelun avulla palvelinympäristöstä voidaan lähettää asiakassovelluksille viestejä. Palvelu perustuu tilaaja-julkaisija-arkkitehtuuriin, jossa julkaisija ei lähetä viestejä tietyille vastaanottajille, vaan viestit kategorisoidaan eri luokkiin ja julkaistaan saataville. Kaikki vastaanottajat, jotka ovat ilmaisseet kiinnostuksensa tietyntyyppisiin viesteihin, vastaanottavat julkaistut viestit. Julkaisija ei siis tiedä, ketkä vastaanottavat

sen julkaisemat viestit.

Tietovarantopalvelu
Tietovarantopalvelun avulla sovellusten palvelinkomponentit voivat tallentaa pysyvästi sovelluskohtaista tietoa relaatiotietokantaan. Palvelu tarjoaa pääsyn sovellusalustasta löytyvän tietokannan sovelluskohtaisiin osiin. Tietovarantopalvelun tarjoamaa tietokanta käytetään JPA-rajapinnan (*Java Persistence API*) kautta.

Hallintapalvelu
Hallintapalvelun avulla sovellusten palvelinkomponentit voivat välittää tietoa järjestelmänhallinnalle ja julkaista tietoa sovelluksen tilasta. Tiedon julkaisu tapahtuu rekisteröimällä sovelluksen määrittämä hallintaolio hallintapalvelulle. Hallintapalvelun kautta tieto välitetään järjestelmänhallinnalle.

Konfiguraatiopalvelu
Konfiguraatiopalvelun avulla sovellusten palvelinkomponentit voivat tallentaa ja hakea sovelluskohtaisia avain-arvo-tyyppisiä asetustietoja. Asetustietoja säilytetään sovellusalustan palvelinympäristössä. Tietojen tallentamisen ja lukemisen lisäksi palvelu tarjoaa mahdollisuuden kuunnella asetustiedoissa tapahtuvia muutoksia.

3.3 Testausjärjestelmän kuvaus

Kehitettävä testausjärjestelmä on tarkoitettu sovellusalustan tarjoamien palvelinympäristön palveluiden suorituskyvyn mittaamiseen. Suorituskykytestausjärjestelmä kehitetään sovellusalustan tuotantokäyttöönottoa varten, ja sen avulla on tarkoitus varmistaa sovellusalustan suorituskyvyn riittävyys suorittamalla sen palveluille kattavat suorituskykytestit. Sovellusalustan omien palveluiden testauksen lisäksi testausjärjestelmän tarkoituksena on tarjota alustaa käyttäville sovelluksille yleiskäyttöinen testauskehys, joka mahdollistaa alustalle kehitettävän sovelluksen suorituskyvyn testaamisen.

3.3.1 Tavoitteet

Suorituskykytestausjärjestelmän tärkeimpänä tavoitteena on mahdollistaa luvussa 3.2 kuvatun tuotteen palvelinympäristön suorituskyvyn testaaminen. Järjestelmällä

on tarkoitus testata sovellusalustan toimintaa, luotettavuutta ja suorituskykyä kuormitettuna, erityisesti tilanteissa, joissa sovellusalustan palvelimelle tehdään runsaasti rinnakkaisia palvelupyyntöjä.

Kehitettävän testausjärjestelmän on tarkoitus mahdollistaa suorituskykytestien ajaminen mahdollisimman vähäisellä työpanoksella. Erityisesti järjestelmän kehittämisellä tavoitellaan tilannetta, jossa yksi testaaja voi suorittaa laajoja, etukäteen määriteltyjä suorituskykytestejä hyvin pienellä työpanoksella, tai täysin automatisoidusti jatkuvan integroinnin yhteydessä. Samalla tavoitteena on myös mahdollistaa tehokas testitulosten tarkastelu ja tarjota valmiita tulosraportteja testien suorituksesta. Koska sovellusalustan kehitys on edelleen kesken, täysin automatisoiduilla suorituskyvyn regressiotesteillä saataisiin lähes välitön palaute kehityksen aikana tehtyjen muutosten vaikutuksesta suorituskykyyn.

3.3.2 Vaatimukset

Testausympäristön tulee pystyä tarjoamaan vertailukelpoisia ja toistettavia suorituskykymittauksia sovellusalustan palvelinympäristöstä. Suorituskykytestausjärjestelmälle oli asetettu seuraavat vaatimukset:

1. Testausjärjestelmän tulee olla helposti siirrettävissä ympäristöstä toiseen.

2. Testausjärjestelmän ajoympäristön tulee vastata lopullisen tuotantoympäristön konfiguraatiota sovelluspalvelimen ja tähän liittyvien oheisohjelmien osalta.

3. Rinnakkaisia käyttäjiä simuloivista asiakassovelluksista tulee olla helposti monistettavissa useita kappaleita useille koneille.

4. Asiakassovelluksien ohjaaminen tulee tapahtua komentorivin kautta.

Ensimmäinen vaatimus on seurausta kohdejärjestelmän käyttöympäristöstä. Koska kohdejärjestelmän tulee olla siirrettävissä ympäristöstä toiseen, sama vaatimus pätee myös sen testausjärjestelmään. Suorituskykytestausjärjestelmällä on tarkoitus mitata järjestelmän tehokkuutta todellisessa tuotantoympäristössä, mikä aiheuttaa vaatimuksen kaksi. Testausjärjestelmän tulee siis olla käytettävissä tuotantoympäristöä vastaavaan ympäristöön viedyn kohdejärjestelmän kanssa, jolloin itse tes-

tauskehys ei saa asettaa ajoympäristölle tätä estäviä vaatimuksia. Kolmas ja neljäs vaatimus ovat seurausta siitä, että testausjärjestelmällä pyritään testaamaan erityisesti kohdejärjestelmän skaalautuvuutta, mutta asiakassovellusten ajoympäristö ei välttämättä sisällä graafista käyttöliittymää.

Lisäksi järjestelmälle oli asetettu seuraavat täydentävät vaatimukset:

1. Testausjärjestelmä tulee olla laajennettavissa siten, että vanhojen testitapausten rinnalle voidaan vaivattomasti lisätä uusia testitapauksia.

2. Kohdejärjestelmän alustallisia palveluita testaavien testitapausten lisäksi testausjärjestelmän tulee mahdollistaa myös kohdejärjestelmää käyttävien sovelluskohtaisten testitapausten rakentamisen ja ajamisen.

Testausjärjestelmän tulee olla siis mahdollisimman tehokkaasti laajennettavissa uusilla testitapauksilla ja sen tulee tarjota sovellusalustan päälle kehitettäville sovelluksille mahdollisuuden kehittää testejä, jotka testaavat itse sovelluksen suorituskykyä. Siten testausjärjestelmä olisi yksi alustan tarjoamista palveluista, joita loppukäyttäjän sovellukset voisivat tarvittaessa hyödyntää. Täydentävien vaatimusten lisäksi testausjärjestelmältä on toivottavaa, että se hyödyntää toteutuksessaan jo valmiita sovellusalustan tarjoamia palveluita. Valmiiksi toteutettuja palveluita käyttämällä testausjärjestelmän kehitys helpottuu ja se saadaan nopeammin käyttöön.

Sovellusalustan arkkitehtuuri aiheuttaa myös testausjärjestelmälle tiettyjä teknisiä vaatimuksia. Koska sovellusalustan ajoympäristö perustuu OSGi-moduulijärjestelmään, myös testausjärjestelmän tulee olla OSGi-yhteensopiva. Lisäksi sovellusalustan kehityksessä on hyödynnetty vahvasti avoimen lähdekoodin ohjelmistoja, joten myös testausjärjestelmän tulee hyödyntää avoimen lähdekoodin ratkaisuja.

3.4 Testaustyökalujen kartoitus

Ennen testausjärjestelmän toteutuksen aloittamista oli tehty kartoitus markkinoilta löytyviin testaustyökaluihin, jotka mahdollisesti täyttäisivät alikohdassa 3.3.2 esitetyt vaatimukset, ja joita voitaisiin käyttää testausjärjestelmän runkona. Sen lisäksi, että työkalun tuli olla avointa lähdekoodia, potentiaalisten työkalujen määrää rajoitti sovellusalustan vaatima OSGi-yhteensopivuus. Testejä tulee voida ajaa sovellusalustan tuotantoympäristöä vastaavalla konfiguraatiolla, joten testausjärjestelmän testitapauksilla tulee olla pääsy OSGi-standardin mukaisesti julkaistuihin

palveluihin vastaavasti kuin todellisessa käyttöympäristössä. Kartoituksen seurauksena oli havaittu, että mahdollisia hyödynnettäviä työkaluja ovat Pax Exam ([9]), JUnit4OSGi ([5]), OSGi-Testrunner ([8]) ja JUnit on OSGi ([4])

Mikään tarkastelluista valmiista ratkaisuista ei kuitenkaan sisältänyt tarvittavia ominaisuuksia. Tarkasteltujen työkalujen ongelmat liittyivät muun muassa testien suorituksen ohjaukseen liittyviin asioihin ja OSGi-ajoympäristön hallintaan liittyviin puutteisiin. Jokaiseen tarkastelluista työkaluista olisi pitänyt tehdä muutoksia, ja tämän vuoksi oman järjestelmän kehittäminen nähtiin parhaana vaihtoehtona, jotta testausjärjestelmä saadaan täysin vaatimusten mukaiseksi.

Testausjärjestelmään toteutettavien suorituskykytestien ajoympäristöksi valikoitui JUnit-testauskehys. JUnit on kehitetty erityisesti yksikkötestauksen tarpeisiin, mutta sen mekanismien nähtiin sopivan myös suorituskykytestien toteutukseen. JUnit valittiin ajoympäristöksi sen tunnettuuden ja yleisyyden vuoksi. Javan käytetyimpänä yksikkötestauskehyksenä sillä on laaja käyttäjäkunta, jolloin suorituskykytestien toteuttajien ei todennäköisesti tarvitse opetella uuden testaustyökalun käyttöä suorituskykytestien luomiseksi.

4. TESTAUSJÄRJESTELMÄN TOTEUTUS

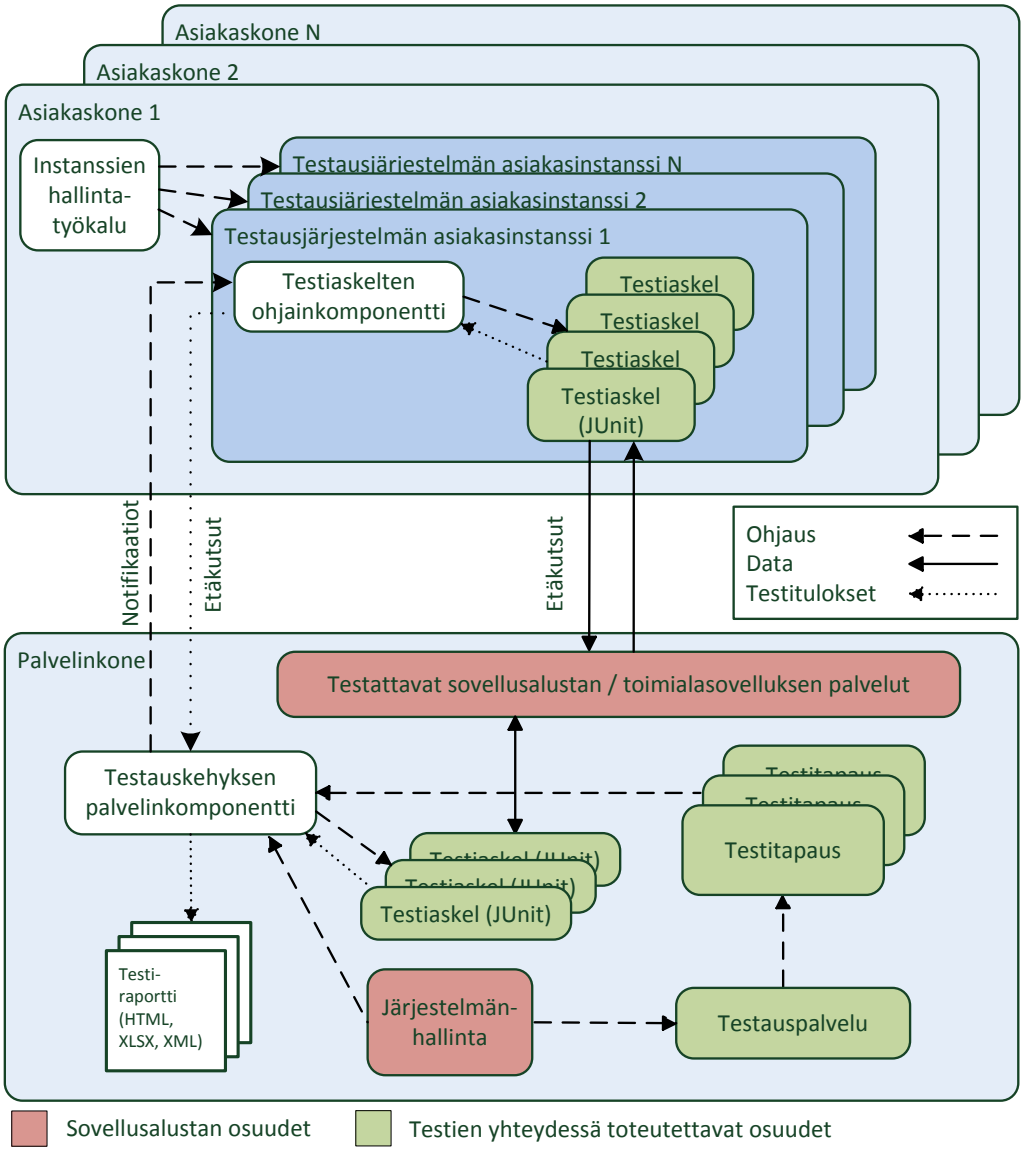
Tässä luvussa kuvataan toteutettu testausjärjestelmä. Aluksi tehdään katsaus järjestelmän korkean tason rakenteeseen, minkä jälkeen järjestelmän palvelin- ja asiakasympäristöjä tarkastellaan tarkemmin. Lopuksi luvussa kuvataan testien määrittelyä ja niiden suoritusta.

4.1 Testausjärjestelmän rakenne

Testausjärjestelmä koostuu sovellusalustan arkkitehtuurin mukaisesti palvelin- ja asiakasympäristöistä. Testausjärjestelmän palvelinosuutta ajetaan sovellusalustan palvelinympäristössä, ja se sisältää testausjärjestelmän ydintoiminnot muodostavan *testauskehyksen palvelinkomponentin*, testien ajon käynnistykseen käytettävän *testauspalvelun*, *testitapaukset* sekä palvelinympäristössä ajettavat *testiaskeleet*. Toteutetun testausjärjestelmän palvelin- ja asiakasympäristöjen korkean tason rakenne on esitetty kuvassa 4.1.

Asiakasympäristö muodostuu kokonaisuudessaan yhdestä tai useammasta fyysisestä *asiakaskoneesta*. Asiakaskoneet ovat yhteydessä palvelinympäristöön verkkoyhteydellä. Yhdellä asiakaskoneella on yksi tai useampi *asiakasinstanssi*. Asiakasinstanssit ajetaan kevyessä OSGi-ajoympäristössä, ja ne koostuvat testiaskelten suorittamiseen käytettävästä *testiaskelten ohjainkomponentista* sekä asiakasympäristön testiaskelista. Lisäksi asiakasinstanssit sisältävät sovellusalustan tarjoamia osuuksia, joilla asiakasinstanssit voivat kommunikoida palvelinympäristön kanssa. Näihin kuuluvat mm. etäkutsujen sekä notifikaatioiden käyttöön liittyvät palvelut. Asiakasinstanssien käynnistys ja pysäytys sekä muut ohjaustoiminnot suoritetaan komentoriviltä käytettävällä *instanssien hallintatyökalulla*.

Testauspalvelu on sovellusalustan järjestelmänhallinnan kautta käytettävä palvelu, jonka avulla testitapaukset käynnistetään ja niille annetaan tarvittavat parametrit. Varsinaiset suorituskykytestit ovat testitapauksia, joiden vaiheet koostuvat tes-



Kuva 4.1 *Testausjärjestelmän rakenne*

tiaskelista. Testitapaus määrittää testin sisältämien testiaskelten suoritusvaiheet ja käyttää testauskehyksen rajapintoja testiaskelten käynnistämiseen, tulosten keräämiseen ja analysointiin. Testiaskelten testitapauksille lähettämät suorituskykymittaukset välitetään testin suorituksen lopuksi testauskehyksen palvelinkomponentille, joka tuottaa mittauksista tietyt tunnusluvut sisältävät testausraportit.

Testausjärjestelmän toiminnallisuuden runkona toimii testauskehys, joka huolehtii testiaskelten käynnistämisestä, kommunikaatiosta muiden järjestelmän osien kanssa sekä tulosten keräämisestä ja analysoinnista. Testauskehys koostuu asiakas- ja palvelinkomponenteista. Testauskehyksen asiakaskomponentti, eli testiaskelten ohjainkomponentti, sisältää nimensä mukaisesti testiaskelten käynnistämiseen ja ohjaukseen tarvittavat luokat. Testauskehyksen palvelinkomponentti sisältää testiaskelten ohjainkomponentin lisäksi tulosraporttien luomiseen ja asiakasinstanssien ohjaukseen liittyvät osuudet.

Testausjärjestelmän asiakas- ja palvelinympäristöt kommunikoivat keskenään verkkoyhteyden avulla käyttäen alikohdassa 3.2.2 kuvattuja etäkutsuja sekä notifikaatioita. Testauskehyksen palvelinkomponentti ohjaa keskitetysti testien suoritusta lähettämällä asiakasympäristön testiaskelten ohjainkomponenteille notifikaatioita. Asiakasinstanssien testiaskeleet kutsuvat palvelinympäristössä sijaitsevia sovellusalustan tai toimialasovelluksen palveluita käyttäen etäkutsuja. Lisäksi testiaskelten ohjainkomponentti lähettää tulokset palvelinkomponentille niin ikään etäkutsulla.

Testausjärjestelmän testien suoritus on toteutettu siten, että testausjärjestelmän eri osat eivät kommunikoi keskenään testiaskelten suorituksen aikana. Kun testiaskeleen suoritus on alkanut, testausjärjestelmän palvelinosuus ei lähetä notifikaatioita testiaskelten ohjainkomponentille, eivätkä ohjainkomponentit lähetä testausjärjestelmän toimintaan liittyviä etäkutsuja. Testiaskelten ohjainkomponentit eivät siis erikseen ilmoita palvelinkomponentille, että testiaskeleen suoritus on valmis, vaan palvelinkomponentti pyytää ohjainkomponenteilta testiaskelten tulokset erikseen määriteltävän ajan kuluttua. Näin eri osuuksien välinen kommunikointi ei aiheuta järjestelmään ylimääräistä kuormaa, mikä vaikuttaisi suorituskykytuloksiin. Tällä pyritään varmistamaan suorituskykymittausten oikeellisuus ja toistettavuus.

Testitapausten varsinainen toiminnallisuus koostuu testiaskelista. Yksi testitapaus koostuu yhdestä tai useammasta testiaskeleesta. Yksittäiset testiaskeleet ovat JUnit-yksikkötestauskehyksen mukaisia luokkia. Yksittäiset testiaskeleet voidaan sijoittaa joko asiakas- tai palvelinympäristössä suoritettaviksi. Näin järjestelmällä voidaan kehittää myös vain palvelinympäristössä ajettavia testejä. Sovellusalustan tai loppukäyttäjäsovelluksen palveluita testattaessa testiaskel kutsuu niiden palveluita. Riippuen kutsuttavan palvelun sijainnista kutsut tehdään joko normaaleina palvelukutsuina tai etäkutsuina verkon yli.

4.2 Asiakasympäristö

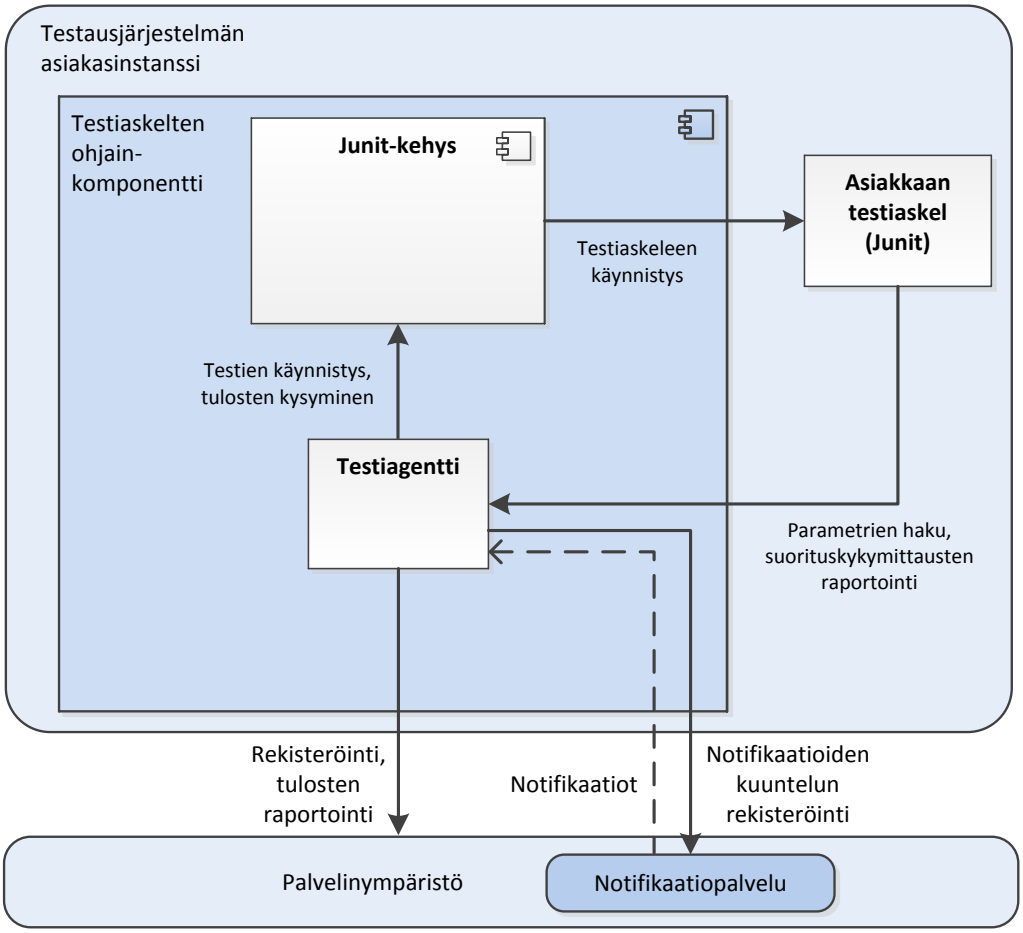
Testausjärjestelmän asiakasympäristön tehtävänä on tuottaa sovellusalustan palvelinkoneelle hallitusti kuormaa. Kuorma saadaan aikaan monistamalla riittävä määrä asiakasinstansseja fyysisille asiakastietokoneille. Nämä asiakasinstanssit simuloivat sovellusalustaa käyttäviä todellisia käyttäjiä, ja niiden tekemät palvelupyynnöt muodostavat palvelinympäristöön varsinaisen kuorman. Palvelupyynnöt tehdään asiakasinstansseilla sijaitsevista testiaskelista. Riittävän kuorman saamiseksi asiakasinstansseja voidaan monistaa asiakaskoneiden määrän ja resurssien puitteissa suuriakin määriä.

Testausjärjestelmän asiakaskokonaisuus koostuu yksittäisestä asennuspaketista, joka voidaan asentaa tietyin rajoituksin mille tahansa yleiskäyttöiselle tietokoneelle, johon on asennettu Java-ajoympäristö sekä tarvittavat varusohjelmat. Asennuspaketti sisältää tarvittavat sovelluspalvelimen osuudet, testiaskelten ohjainkomponentin, asiakasympäristön testiaskeleet sekä asiakasinstanssien hallintatyökalun. Asiakasinstansseja ajetaan Apache Karaf -OSGi-ajoympäristössä. Yksittäisten asiakasinstanssien viemien resurssien määrää on pyritty minimoimaan, jotta niitä voidaan monistaa yhdelle asiakaskoneelle mahdollisimman monta ilman, että koneen muisti ja suoritusteho loppuvat.

4.2.1 Rakenne ja tehtävät

Testien käynnistykseen ja ohjaukseen liittyvä logiikka sijaitsee testiaskelten ohjainkomponentissa. Ohjainkomponentti sisältää *testiagentin* ja *JUnit-kehyksen*. Kuva 4.2 esittää yksittäisen asiakasinstanssin rakennetta. Testiagentti on ohjainkomponentin ydin ja sen tehtävät ovat:

- asiakasinstanssin rekisteröinti testauskehyksen palvelinkomponentille instanssin käynnistyksen yhteydessä
- testauskehyksen palvelinkomponentin lähettämien notifikaatioiden kuuntelu ja niihin reagointi
- testiaskelten käynnistys JUnit-kehyksen kautta
- tulosten pyytäminen JUnit-kehykseltä



Kuva 4.2 *Testiaskelten ohjainkomponentin rakenne*

- JUnit-kehykseltä ja testiaskelilta saatujen tulosten raportointi testausjärjestelmän palvelinkomponentille.

JUnit-kehys sisältää luokkia, joilla JUnit-testejä voidaan ajaa OSGi-ympäristössä. JUnit-kehyksen tehtävänä on säilyttää tieto ajettavista testiaskelista. Kehyksen vastuulla on myös yksittäisen testiaskeleen käynnistäminen ja keskeyttäminen sekä testiaskelten suorituksen onnistumisen raportointi.

Testiaskel koostuu yksittäisestä JUnit-testiluokasta, ja sen tehtävänä on kuorman luominen palvelinympäristölle palvelukutsujen avulla. Testiaskel myös huolehtii suo-

rituskykymittausten tekemisestä sekä mittaustulosten raportoinnista testiagentille.

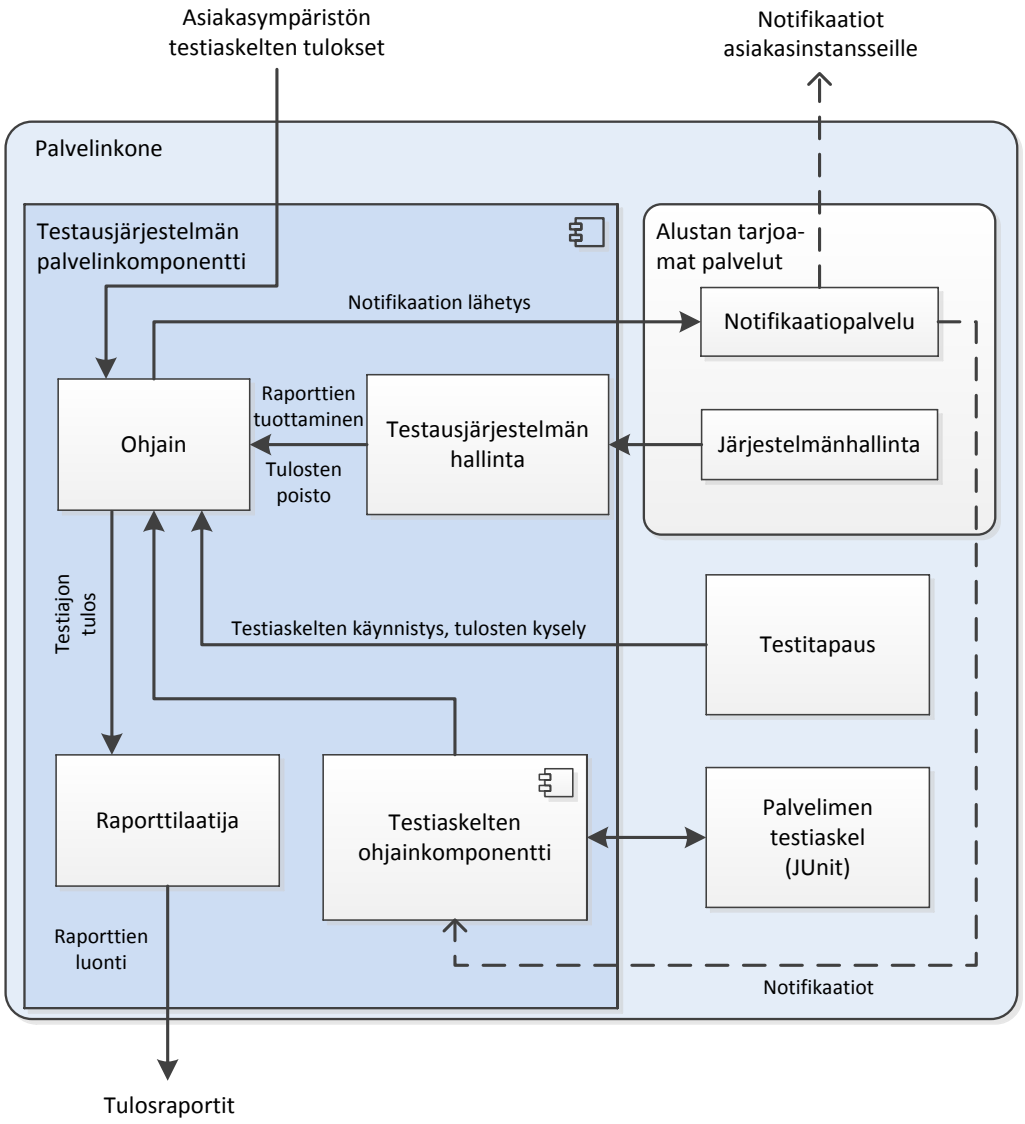
4.2.2 Toimintaperiaate

Asiakasinstanssin käynnistyessä testiagentti lähettää tiedon käynnistyneestä instanssista rekisteröimällä itsensä palvelinkomponentille. Samalla testiagentti rekisteröityy sovelluspalvelimen notifikaatiopalvelulle vastaanottamaan testausjärjestelmän palvelinkomponentin lähettämiä ohjausnotifikaatioita. Tämän jälkeen testiagentti jää odottamaan mahdollisia palvelimelta tulevia komentoja. Notifikaation saapuessa testiagentti tekee notifikaation sisältämät toimenpiteet. Mikäli notifikaation tyyppinä on testiaskeleen käynnistys, pyyntö delegoidaan JUnit-kehykselle, joka käynnistää testiaskeleen ajon. Jos notifikaation tyyppi on tulosten keräys, testiagentti lähettää testiaskelten ja JUnit-kehyksen raportoimat testitulokset palvelinkomponentille.

Testiaskeleen käynnistysnotifikaation saapuessa JUnit-kehys käynnistää testiaskeleen testiagentin kutsusta. Testiaskeleen ajon päättyessä JUnit-kehys säilöö tiedon testin onnistumisesta tai mahdollisesta epäonnistumisesta sekä koko testiaskeleen suoritukseen kuluneesta ajasta. Testiaskeleen vastuulla on varsinaisiin palvelukutsuihin liittyvien tulosten ja mittausten kerääminen. Tällainen voi olla esimerkiksi jonkin tietokantaoperaation suorittamiseen kulunut aika. JUnit-kehyksen keräämät tulokset kertovat siis pääpiirteittäin testiaskeleen onnistumisesta testiaskeleen keräämien varsinaisten tulosten ollessa sovellusalakohtaisia suorituskyvyn mittaustuloksia. Kun testiaskeleen suoritus on päättynyt, testiaskel lähettää suorituskykymittaustulokset testiagentille. Testiagentti lähettää JUnit-kehyksen ja testiaskeleen keräämät tulokset palvelinkomponentille, kun testiaskeleen raportointinotifikaatio saapuu testiagentille.

4.3 Palvelinympäristö

Testausjärjestelmän palvelinympäristön tehtävänä on käynnistää testit ja ohjata keskitetysti koko testausjärjestelmän toimintaa. Testien käynnistäminen tapahtuu palvelimella sijaitsevan testauspalvelun kautta, joka käynnistää yksittäiset testitapaukset. Testitapaukset ohjaavat yksittäisten testiaskelten suoritusta kutsumalla testauskehyksen palvelinkomponentin rajapintoja.



Kuva 4.3 *Testauskehyksen palvelinkomponentin rakenne*

4.3.1 Rakenne ja tehtävät

Testausjärjestelmän palvelinkomponentti sisältää kaiken testausjärjestelmän ohjaamiseen tarvittavan ohjelmalogiikan. Testausjärjestelmän palvelinkomponentti sisältää *ohjaimen*, järjestelmänhallinnan kautta käytettävän *testausjärjestelmän hallinnan*, testien tulosraportit laativan *raporttilaatijan* sekä myös asiakasympäristössä sijaitsevan *testiaskelten ohjauskomponentin*. Testausjärjestelmän palvelinkomponentin rakenne on esitetty kuvassa 4.3.

Ohjain sisältää ydintoiminnot testausjärjestelmän ja asiakasinstanssien ohjaamiseen, ja sen tehtävät ovat:

- rekisteröityneiden testiagenttien tietojen ylläpito
- testiagenttien ohjaaminen notifikaatioita lähettämällä
- tulosten kysely testiagenteilta
- testiaskelten tulosten säilöminen
- testiaskelten tulosten lähettäminen testitapaukselle
- testitapausten kokonaistulosten tallentaminen ja lähettäminen raportinlaatijalle.

Testausjärjestelmän hallinta on sovellusalustan järjestelmänhallinnan kautta käytettävä palvelu, joka huolehtii testausjärjestelmään kirjautuneiden testiagenttien määrän ja testien suorituksen etenemisestä kertovan juoksevan lokitulosteen näyttämisestä. Lisäksi testausjärjestelmän hallinnan tehtävänä on lähettää pyyntö ohjaimelle tulosraporttien tuottamiseksi ja tarvittaessa poistaa ohjaimella sijaitsevat vanhat testitulokset.

Raporttilaatija sisältää toiminnallisuuden valmiiden tulosraporttien luomiseen testien ajojen perusteella. Raporttilaatijan vastuulla on tiettyjen tunnuslukujen laskeminen testitulosten mittausarvoista sekä tulosraporttien luominen ja tallentaminen levyjärjestelmään.

Testiaskelten ohjainkomponentin tehtävät ovat vastaavat kuin asiakasympäristössä. Testiaskelten ohjainkomponentti sijaitsee testausjärjestelmän palvelinkomponentissa palvelinympäristössä ajettavia testiaskelien suoritusta varten.

4.3.2 Toimintaperiaate

Testausjärjestelmän palvelinkomponentissa sijaitseva ohjain pitää kirjaa testausjärjestelmään rekisteröityneistä testiagenteista. Kun testiagentin sisältävä testiaskelten ohjainkomponentti käynnistetään joko asiakasinstanssissa tai palvelinympäristössä,

testiagentti rekisteröityy ohjaimelle ja sovellusalustan notifikaatiopalvelulle kuuntelemaan ohjaimen lähettämiä ohjausnotifikaatioita.

Ohjain huolehtii yksittäisiin testiaskeliin liittyvien ohjaustoimien tekemisestä. Testien kokonaisuuden hallinta tapahtuu testitapausluokassa, jossa määritellään testiaskelten suoritusjärjestys ja askelten välissä odotettava aika. Testitapausten tehtäviä ja määrittelyä tarkastellaan tarkemmin alikohdassa 4.4.2. Testitapausluokasta kutsutaan ohjaimen rajapintaa joko testiaskeleen käynnistämiseksi tai tulosten keräämiseksi. Kutsun saapuessa ohjain julkaisee notifikaation testiaskeleen käynnistämiseksi tai tulosten keräämiseksi kaikille vastaanottajille, jotka ovat rekisteröityneet kuuntelemaan ohjaimen lähettämiä notifikaatioita.

Testausjärjestelmän hallinta on sovellusalustan järjestelmänhallinnalle rekisteröity hallintaolio, jolla testausjärjestelmän palvelinkomponenttia voidaan ohjata. Testausjärjestelmän hallinta sisältää juoksevan lokitulostuksen testien etenemisestä, ja sen kautta voidaan tuottaa valmiita tulosraportteja ajetuista testeistä sekä poistaa ohjaimella sijaitsevat testitulokset. Kun testausjärjestelmän hallinnan kautta tulee pyyntö tuottaa tulosraportit, ohjain lähettää tallentamansa testitapausten tulokset raporttilaatijalle. Raporttilaatija tuottaa saaduista testituloksista tulosraportit, jotka se tallentaa levyjärjestelmään. Raporttilaatijan tuottamia tulosraportteja tarkastellaan tarkemmin kohdassa 5.3.

4.4 Testien määrittely

Testausjärjestelmään toteutettujen suorituskykytestien kokonaisuus sisältää testauspalvelun, testitapauksia ja testiaskelia. Testauspalvelu on sovellusalustan web-pohjaisen järjestelmänhallinnan kautta käytettävä palvelu, josta testitapaukset parametrisoidaan ja käynnistetään. Testikokonaisuuksien toteutuksen yhteydessä voidaan määritellä tarvittaessa useita testauspalveluita. Testauspalvelun toiminnallisuus määritellään käyttäen erityistä sovellusalustan palvelukuvausta, mutta sen määrittely jätetään tämän työn ulkopuolelle.

Testausjärjestelmä mahdollistaa myös monen testitapauksen suorittamisen peräkkäin käyttäen sekvenssejä. Testiaskelten, testitapausten ja sekvenssien määrittelyä tarkastellaan seuraavaksi tarkemmin. Seuraavissa kohdissa käytetään esimerkkinä testiä, joka testaa suorituskykytestejä varten toteutetun etäkutsurajapinnan toimintaa. Testissä asiakasympäristön testiagenttien tarkoituksena on kutsua toistu-

vasti palvelinympäristössä sijaitsevan testipalvelun rajapintaa. Esimerkkitesti koostuu testitapauksesta ja yhdestä testiaskeleesta.

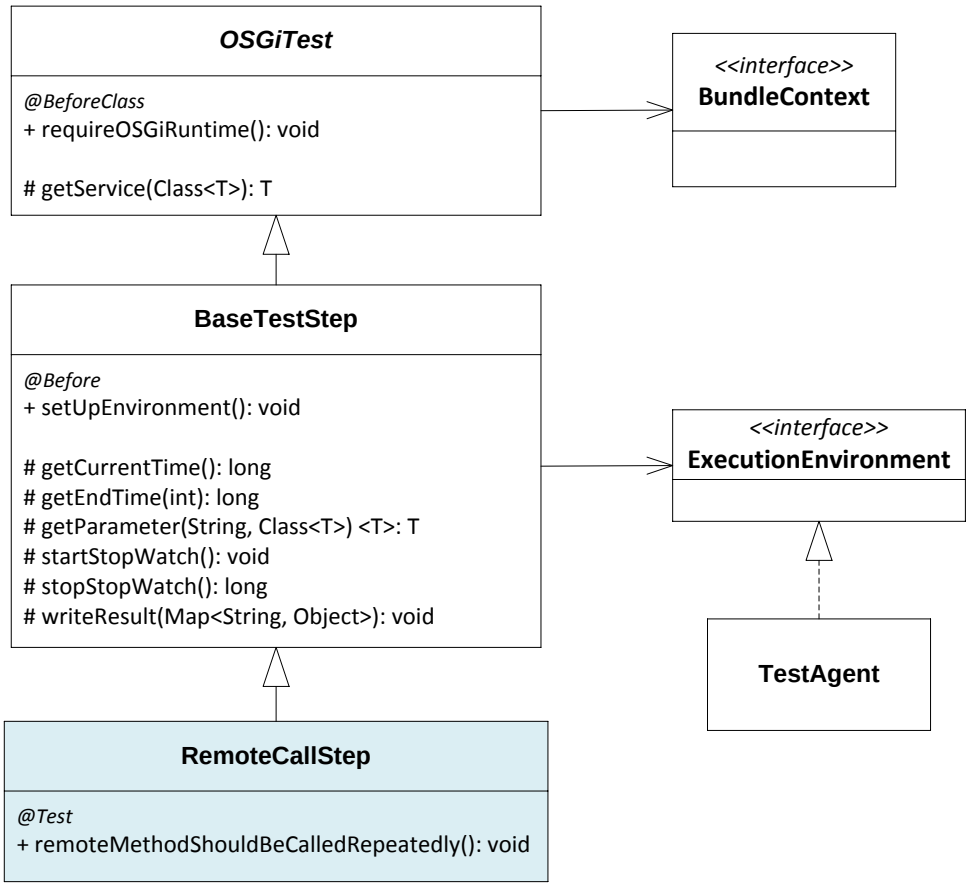
4.4.1 Testiaskeleet

Testiaskeleet ovat JUnit-luokkia, jotka sisältävät testitapausten varsinaisen mittauksen tekevän toiminnallisuuden. Testiaskeleet eivät ole normaaleja JUnit-testejä, vaan JUnit-kehystä käytetään suorituskykytestien ajoympäristönä, ja JUnitin mekanismeja käytetään testien suoritukseen. Testiaskelet voivat hyödyntää kaikkia JUnitin tarjoamia ominaisuuksia. Esimerkiksi yksittäinen testiaskel voidaan asettaa epäonnistuneeksi samaan tapaan kuin normaaleissa yksikkötesteissä käyttämällä assertioita.

Testausjärjestelmän varaan toteutettujen testiaskelten tulee sisältää täsmälleen yksi `@Test`-annotoitu metodi. Tämä metodi sisältää toimenpiteet, joissa kutsutaan testattavan palvelun rajapintoja, mitataan toimenpiteisiin kulunut aika ja tallennetaan sekä lähetetään mittaustulokset testiagentille. Testausjärjestelmä tarjoaa testiaskelille kantaluokkahierarkian, joka sisältää erilaisia metodeita testiagentin kanssa kommunikointiin sekä apufunktioita parametreihin ja ajanottoon liittyen. Testiaskelten luokkakaavio on esitetty kuvassa 4.4.

Koska sovellusalustan ja testausjärjestelmän ajoympäristöt perustuvat OSGiin, ja testiaskelista on tarkoitus kutsua palvelinympäristön palveluita, tulee testiaskelilla olla pääsy palvelinympäristöstä julkaistuihin OSGi-palveluihin. Tämän toiminnallisuuden tarjoaa testausjärjestelmän `OSGiTest`-JUnit-luokka, joka myös tarkistaa ennen ensimmäisen testiaskeleen suoritusta, että testiä ajetaan OSGi-ympäristössä. Mikäli näin ei ole, testejä ei ajeta, koska suurin osa testiaskeleista vaatii OSGi-palveluita suorituskykymittausten toteuttamiseksi.

`OSGiTest`-luokasta periytyvä `BaseTestStep` sisältää testien ajoon liittyviä apufunktioita. Se tarjoaa lisäksi pääsyn testiagentin rajapintaan, jonka avulla testiagentilta voidaan kysyä testiaskeleen suoritukseen tarvittavia parametreja sekä lähettää testiaskeleen mittaustulokset. Lisäksi luokka tarjoaa erilaisia apufunktioita ajan mittaamiseen liittyen. Ote kuvassa 4.4 näkyvästä `RemoteCallStep`-esimerkkitestiaskeleen testimetodista on esitetty ohjelmassa 4.1.



Kuva 4.4 *Testiaskelten luokkakaavio.* `RemoteCallStep` *on esimerkkitoteutus testiaskeleesta.*

Ohjelma 4.1 *Ote esimerkkitestiaskeleen toteutuksesta*

```
1  @Test
2  public void remoteMethodShouldBeCalledRepeatedly() {
3    final Map<String, Object> result = new TreeMap<>();
4    int callCounter = 0;
5    final int testDuration = getParameter("test.duration", int.class);
6    final long endTime = getEndTime(testDuration);
7    while (getCurrentTime() <= endTime) {
8      result.put("remote.call.delay." + callCounter, measureRemoteCallTime());
9      callCounter++;
10   }
11   result.put("remote.call.delays.count", callCounter);
12   writeResult(result);
13 }
```

Esimerkkitestiaskeleen tehtävänä on kutsua etäkutsumekanismilla toistuvasti palvelinpäässä sijaitsevaa palvelua ja tallentaa mittaustuloksina kutsujen kokonaismäärä sekä yksittäisiin kutsuihin kulunut aika. Testiaskeleen riveillä 5 ja 6 testiagentilta haetaan testille parametrisoitu kesto sekä testin loppuaika. Riveillä 7–9 tehdään varsinainen mittaus, ja tallennetaan tulokset. Lopuksi riveillä 11 ja 12 tallennetaan tehtyjen etäkutsujen kokonaismäärä sekä lähetetään testiaskeleen tulokset testiagentille. Testipalvelun kutsun ja ajanoton sisältävä `measureRemoteCallTime`-metodi sekä testiaskeleen muut osat on esitetty kokonaisuudessaan liitteessä 1.

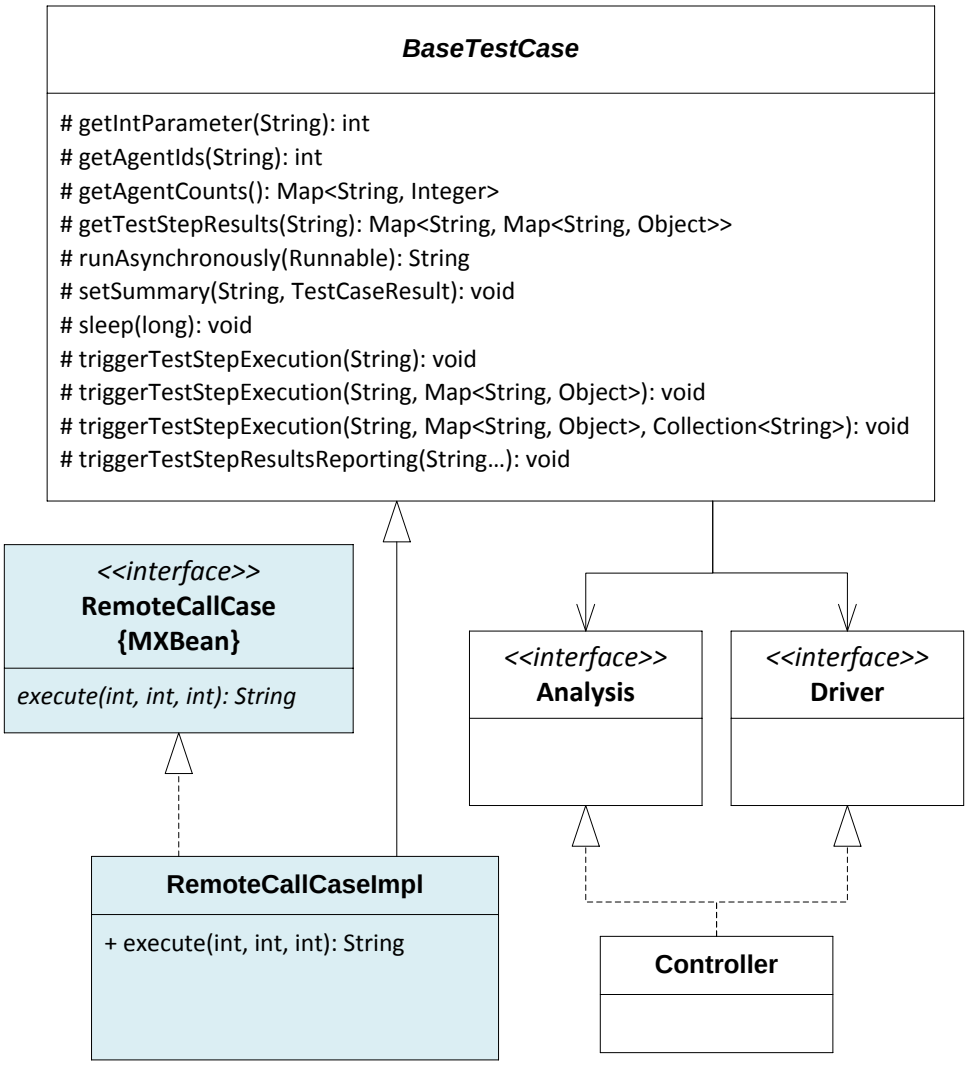
4.4.2 Testitapaukset

Testitapaukset ovat järjestelmänhallinnalle rekisteröitäviä hallintaolioita, joita käytetään testauspalvelun kautta. Testitapaukset koostuvat MBean-annotoidusta (*Managed Bean*) Java-rajapinnasta, jolla määritellään järjestelmänhallinnasta käynnistettävä metodi sekä rajapinnan toteuttavasta Java-luokasta, jossa testin suoritus tapahtuu. Testitapausten tehtävänä on hallita testin etenemistä testauskehyksen palvelinkomponentissa sijaitsevan ohjaimen rajapintojen kautta, testitulosten rakentaminen yksittäisten testiaskelten tulosten perusteella sekä rakennetun testituloksen lähettäminen testauskehyksen palvelinkomponentille.

Testauskehys tarjoaa toteutettaville testitapauksille kantaluokan, johon on kerätty yleisiä testitapauksissa tarvittavia toimintoja. Kantaluokka tarjoaa metodeita muun muassa testiaskelten käynnistämiseen sekä parametrien ja testiagenttien käsittelyyn. Kantaluokan ja siihen liittyvien luokkien luokkakaavio on esitetty kuvassa 4.5.

Kantaluokka kutsuu metodeistaan testauskehyksen palvelinkomponentin ohjaimen (`Controller`) tarjoamia rajapintoja, joiden kautta kantaluokka ohjaa testien suoritusta ja pyytää testiaskelten tuloksia. Kantaluokan avulla voidaan muun muassa käynnistää testi asynkronisesti, laukaista testiaskelten suoritus ja tulosten raportoiminen sekä kysyä tietoa testausjärjestelmään rekisteröityneistä testiagenteista. Lisäksi kantaluokka tarjoaa metodit konfiguraatioparametrien arvojen ja testiaskelten tulosten hakemiseen sekä testin kokonaistuloksen lähettämiseen ohjaimelle.

Kantaluokka tarjoaa testiaskelten suorituksen laukaisuun kuormitetun `triggerTestStepExecution`-metodin, jonka avulla testiaskelia voidaan käynnistää eri tavoilla. Testiaskeleen käynnistys voidaan laukaista kaikille rekisteröityneille testiagenteille testiaskeleen tunnisteen (`String`) perusteella ilman parametreja ja para-



Kuva 4.5 *Testitapausten luokkakaavio.* `RemoteCallCase` *ja* `RemoteCallCaseImpl` *muodostavat erään toteutuksen testitapauksesta.*

metrien (`Map`) kanssa. Lisäksi metodille voidaan antaa lista (`Collection`) testin suorittavista testiagenteista, jolloin testiaskel suoritetaan vain listan mukaisilla testiagenteilla. Näin testiaskeleen suorittavien testiagenttien määrää voidaan vaihdella eri testiaskelten välillä.

Kuvassa 4.5 esitetty `RemoteCallCase`-esimerkkitestitapaus on määritelty testauspalveluun siten, että testin suoritus tapahtuu `execute`-metodissa. Ote esimerkkites-

titapauksen toteutuksesta on esitetty ohjelmassa 4.2.

Ohjelma 4.2 *Ote esimerkkitestitapauksen toteutuksesta*

```
1  @Override
2  public String execute(final int parameterAmount, final int parameterSize,
       final int testDuration) {
3    return runAsynchronously(new Runnable() {
4
5      @Override
6      public void run() {
7        final Map<String, Integer> agentCounts = getAgentCounts();
8        final Map<String, Object> testStepParameters = new TreeMap<>();
9        testStepParameters.put("parameter.count", parameterAmount);
10       testStepParameters.put("parameter.size", parameterSize);
11       testStepParameters.put("test.duration", testDuration);
12       triggerTestStepExecution("com.example.RemoteCallStep",
             testStepParameters);
13       sleep(getIntParameter("short.safe.delay") + 1000 * testDuration +
             getIntParameter("max.iteration.delay"));
14       triggerTestStepsResultsReporting("com.example.RemoteCallStep");
15       sleep(getIntParameter("max.report.delay"));
16       analyze(agentCounts, parameterAmount, parameterSize, testDuration);
17     }
18   });
19 }
```

Esimerkkitoteutuksen rivillä 2 määritelty `execute`-metodi suoritetaan, kun testi käynnistetään testauspalvelusta. Testin suoritus alkaa agenttien lukumäärän hakemisella ja parametrien käsittelyllä (rivit 7-11). Agenttien lukumäärää käytetään myöhemmin testituloksen luomiseen, ja parametrit lähetetään testiaskeleen suorittaville testiagenteille. Ensimmäinen testiaskel käynnistetään rivillä 12, jossa määritellään käynnistettävän testiaskeleen luokkapolku (FQDN, engl. *Fully qualified domain name*) ja testiaskeleen parametrit.

Askeleen käynnistyksen jälkeen testiä suorittava säie asetetaan odottamaan `sleep`-metodille parametrina annetuksi ajaksi, joka esimerkin tapauksessa on testin kesto millisekunteina lisättynä konfiguraatiopalvelusta haetun konfiguraatioarvon mukaisella ajalla. Testiaskelten välissä odotettava aika riippuu testiaskeleen sisältämästä toiminnallisuudesta, ja sen tarkoituksena on varmistaa, että kaikki testiagentit ehtivät suorittaa testiaskeleen loppuun asti. Odotuksen jälkeen testiagenteilta kysytään

testiaskeleen tulokset (rivi 14). Mikäli testiaskeleen jälkeinen viive on asetettu liian lyhyeksi, ja yksi tai useampi testiagentti ei ole ehtinyt suorittaa testiaskelta loppuun asti, niiden tulokset eivät ole vielä saatavilla, jolloin testiaskeleen tulokset eivät ole luotettavia. Tulosten kysymisen jälkeen odotetaan riittävä aika, jotta testiagentit ehtivät raportoida tulokset (rivi 15) sekä tehdään tuloksille analyysi (rivi 16). Metodin `analyze` toteutus sekä testitapauksen muut osat on esitetty kokonaisuudessaan liitteessä 2.

Tulosten analysoinnissa ohjaimelta pyydetään testiaskelten tulokset ja niiden perusteella rakennetaan testin kokonaistulos. Testituloksia kuvaavat `TestCaseResult`-luokan oliot. Nämä oliot koostuvat testin parametreja, epäonnistumisia ja mittaustuloksia kuvaavista `Parameter`-, `Failure`- ja `Measurement`-luokkien olioista.

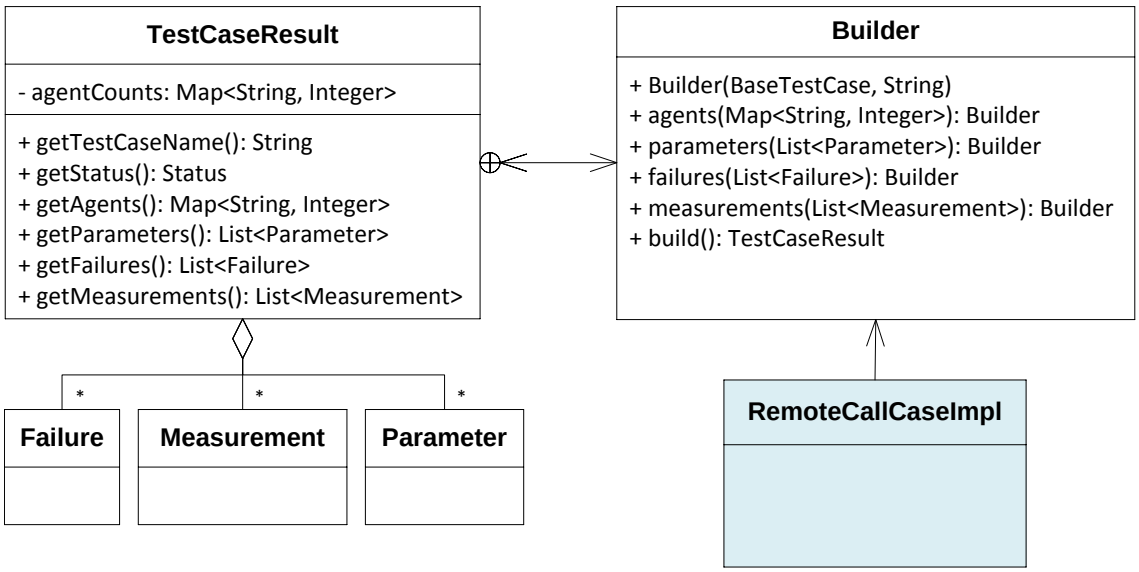
Tulosten analysoinnin yhteydessä testiaskelten tekemistä mittaustuloksista luodaan `Measurement`-luokan olioita, jotka esitetään tulosraporteissa mittaustuloksina. Measurement-olio sisältää tulosraportissa esitettävän mittaustuloksen nimen sekä yhden tai useamman numeerisen mittausarvon. Esimerkki etäkutsujen kestojen mittausarvoja sisältävästä Measurement-olion luomisesta on esitetty alla:

```
Measurement measurement = new Measurement("Remote call delay");
Integer callCount = (Integer) result.get("remote.call.delays.count");
for (int i = 0; i < callCount; i++) {
  Long callDelay = (Long) result.get("remote.call.delay." + i);
  measurement.addValue(callDelay);
}
```

Testitulos rakennetaan `TestCaseResult`-luokan sisäisellä `Builder`-rakentajaluokalla. Luokat `TestCaseResult` ja `Builder` on esitetty kuvassa 4.6. Alla on esimerkki testituloksen rakentamisesta `Builder`-luokalla:

```
final Builder builder = new Builder(this, "Remote call test");
builder.agents(agentCounts);
builder.parameters(parameters);
builder.failures(failures);
builder.measurements(measurements);
setSummary("Remote call test", builder.build());
```

Testituloksen rakentajalle annetaan instantioinnin yhteydessä testitapausluokan ilmentymä sekä testin nimi. Tämän jälkeen rakentajalle annetaan testin suorittanei-



Kuva 4.6 *Testitulosten luokkakaavio*

den testiagenttien lukumäärät, testin parametrit, epäonnistuneet testiaskeleet sekä testin varsinaiset mittaustulokset. Lopuksi kutsutaan kantaluokan `setSummary`-metodia, joka lähettää tuotetun testituloksen testauskehyksen ohjaimelle.

4.4.3 Sekvenssit

Testausjärjestelmä mahdollistaa testien ajamisen peräkkäin testisekvensseinä. Testisekvenssien avulla voidaan muodostaa pitkiä testiajoja, jotka koostuvat yhdestä tai useammasta testitapauksesta eri parametriyhdistelmillä. Testisekvenssit ovat XML (engl. *Extensible Markup Language*)-kuvauskielellä toteutettuja tiedostoja, joissa on listattu ajettavat testit ja niiden parametrit. Sekvenssien XML-skeema sisältää mahdollisuuden seuraaviin toimintoihin:

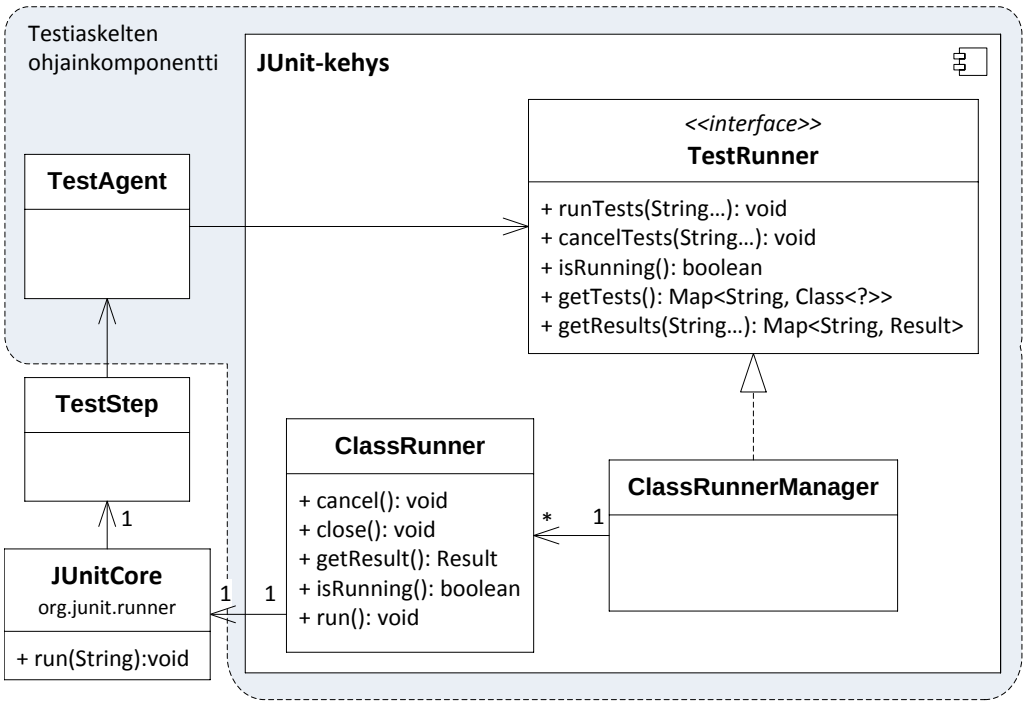
- testitapauksen parametrisointi ja käynnistäminen
- viiveen määrittäminen testitapausten välille
- tulosraporttien generointi
- testitulosten poistaminen.

Ohjelmassa 4.3 on esitetty esimerkki sekvenssistä, joka sisältää kaksi testiajoa. Sekvenssin määrittely alkaa XML-prologin jälkeen `sequence`-elementillä riviltä 2. Testiajojen määritys alkaa riviltä 4, jossa määritellään *Remote call test* -testin käynnistys `case`-elementillä. Elementille annetaan attribuuttina testin käynnistävän metodin nimi. Riveillä 5–8 määritellään testiajolle parametrit `parameter`-elementeillä, joille annetaan attribuutteina parametrin nimi ja arvo. Rivit 11–16 sisältävät vastaavan testiajon määrittelyn eri parametrien arvoilla. Rivillä 18 generoidaan tulosraportit ja lopuksi rivillä 19 testien tulokset poistetaan palvelimen muistista.

Ohjelma 4.3 *Esimerkki testisekvenssistä*

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <sequence xmlns="http://xmlns.insta.fi/2013/1/test-sequence">
3
4    <case name="Remote call test" method="execute">
5      <parameter name="parameterAmount" value="5" />
6      <parameter name="parameterSize" value="10" />
7      <parameter name="testDuration" value="60" />
8    </case>
9    <sleep duration="157000" />
10
11   <case name="Remote call test" method="execute">
12     <parameter name="parameterAmount" value="5" />
13     <parameter name="parameterSize" value="20" />
14     <parameter name="testDuration" value="60" />
15   </case>
16   <sleep duration="157000" />
17
18   <summary />
19   <clear />
20 </sequence>
```

Testisekvenssin toiminnot suoritetaan siinä järjestyksessä, jossa ne on määritelty. Näin testiajoja voidaan luokitella kokonaisuuksiin esimerkiksi testien tyyppien tai parametrien arvojen perusteella. Testiajojen väliin tulee määritellä viive, jonka aikana testiajo saadaan suoritettua. Testausjärjestelmä tarjoaa myös Python-kielellä toteutetun komentorivityökalun, jolla testisekvenssien generointia voidaan helpottaa. Sen tarkastelu jää kuitenkin tämän työn ulkopuolelle.



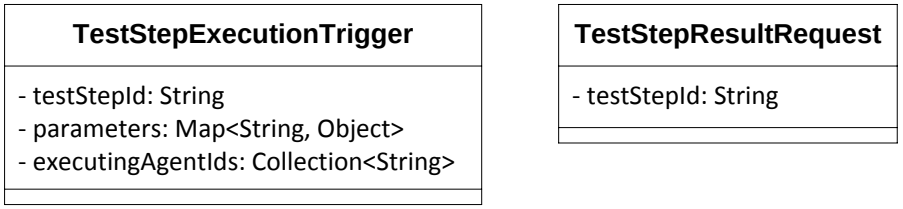
Kuva 4.7 *JUnit-kehyksen luokkakaavio*

4.5 Testien suoritus

Tässä kohdassa tarkastellaan yksittäisen testin suoritusta alusta loppuun kaikkien testausjärjestelmän esiteltyjen osien kannalta. Kohdassa käytetään esimerkkitestinä samaa etäkutsurajapintaa testaavaa testiä kuin kohdassa 4.4. Testitapauksen suoritus koostuu testiaskelten suorituksesta, testiaskelten tulosten keräämisestä ja testin kokonaistuloksen rakentamisesta.

4.5.1 Testiaskelten suoritus ja tulosten kerääminen

Testiaskelten suoritus ja tulosten kerääminen tapahtuvat testiaskeleen ohjainkomponentissa sijaitsevan JUnit-kehyksen avulla. JUnit-kehys ja testiaskeleen ohjainkomponentti sijaitsevat sekä palvelin- että asiakasympäristössä. JUnit-kehys tarjoaa toiminnallisuudet testausjärjestelmän suorituskykytestejä kuvaavien JUnit-testiluokkien ajamiseen, keskeyttämiseen ja tulosten keräämiseen. JUnit-kehyksen luokkakaavio on esitetty kuvassa 4.7.



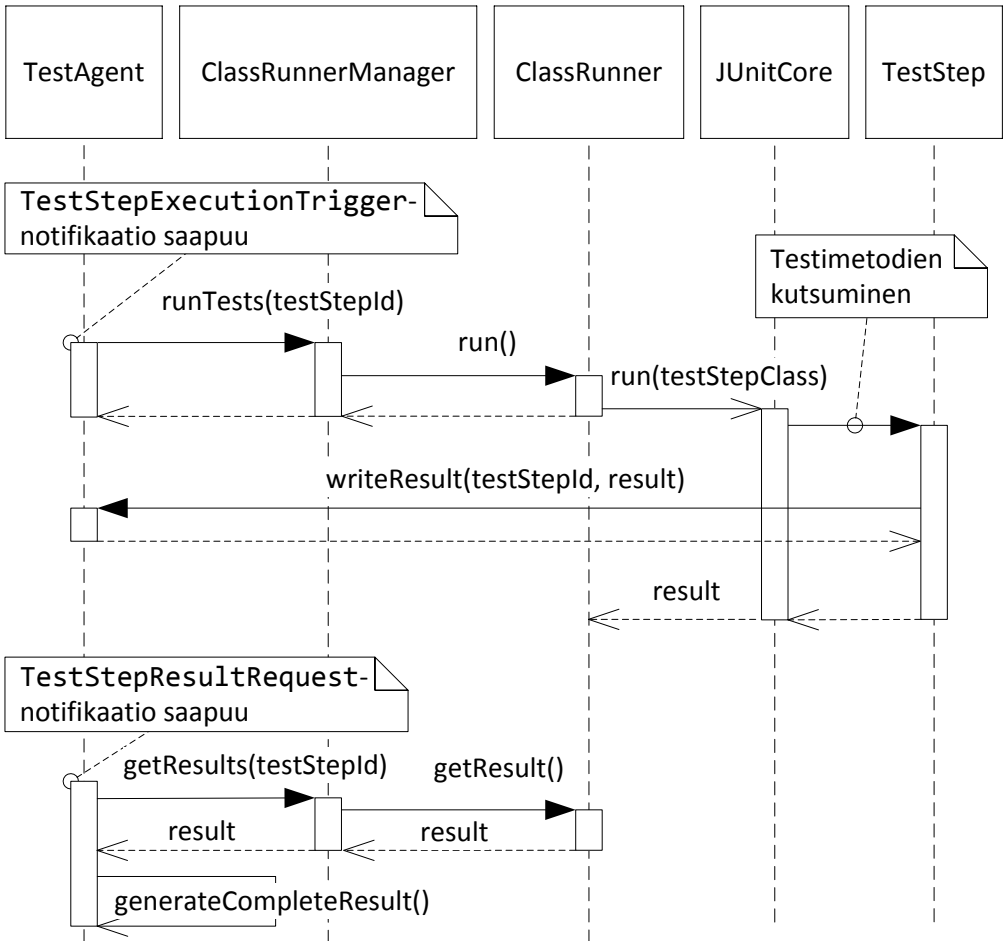
Kuva 4.8 *Notifikaatiopalvelun kautta lähetettävät notifikaatiot*

Testiaskeleen ohjainkomponentin testiagentti (`TestAgent`) käyttää `TestRunner`-rajapintaa JUnit-kehyksen kanssa kommunikointiin. Rajapinnan kautta testiagentti voi käynnistää ja keskeyttää testiaskelia ja kysyä erilaisia tietoja testiaskelten tilasta. Kehyksen `ClassRunnerManager` hallinnoi `ClassRunner`-luokan olioita, joiden tehtävänä on hallita yksittäisen testiaskeleen suoritusta. Jokaiselle ympäristössä olevalle testiaskeleelle luodaan oma `ClassRunner`-luokan ilmentymä. Luokka tarjoaa metodeita testiaskeleen käynnistämiseen, keskeyttämiseen sekä tulosten kysymiseen. JUnit-testien varsinaisen ajoympäristön tarjoaa JUnitin oma `JUnitCore`-luokka, joka huolehtii testiaskeleen JUnit-annotoitujen metodien kutsumisesta ja tuottaa JUnit-testin tuloksen.

Testiaskelten suorituksen laukaisuun ja tulosten keräämiseen käytetään sovellusalustan tarjoaman notifikaatiopalvelun kautta lähetettäviä notifikaatioita. Notifikaatiotyyppejä on kaksi, ja niiden rakenteet on esitetty kuvassa 4.8. Testiaskeleen käynnistävä `TestStepExecutionTrigger`-notifikaatio sisältää testiaskeleen tunnisteen, testiaskeleen parametrit ja listan testiaskeleen suorittavista testiagenteista. Testiaskeleen raportoinnin käynnistävä `TestStepResultRequest`-notifikaatio sisältää vain testiaskeleen tunnisteen, jolta tulokset tulee kerätä.

Testiaskeleen suoritus koostuu seuraavista vaiheista:

1. notifikaation lähettäminen testiagenteille
2. testiaskeleen suorittaminen JUnit-kehyksessä
3. testiaskeleen tulosten raportoiminen testiagentille
4. testiaskeleen suoritukseen liittyvän tuloksen tallentaminen JUnit-kehykseen.



Kuva 4.9 *Sekvenssikaavio testiaskeleen suorituksesta*

Testiaskeleen suoritus alkaa, kun testitapausluokka kutsuu testausjärjestelmän ohjaimen rajapintaa testiaskeleen suorittamiseksi. Kutsun yhteydessä ohjain julkaisee sovellusalustan notifikaatiopalvelun kautta `TestStepExecutionTrigger`-notifikaation rekisteröityneille testiagenteille. Vastaanotettuaan notifikaation testiagentit käynnistävät testin suorituksen, mikäli testiaskel on saatavilla ja notifikaation mahdollisesti sisältämä testiagenttien lista sisältää kyseisen testiagentin. Mikäli lista on tyhjä, kaikki testiagentit suorittavat testiaskeleen. Jos testiaskeleen suoritus aloitetaan, se etenee kuvassa 4.9 esitetyn sekvenssikaavion mukaisesti.

Kutsuketju testiaskeleen käynnistämiseksi etenee `ClassRunner`-luokalle, joka asettaa testiaskeleen käynnistettäväksi asynkronisesti. Testiaskeleen suorituksen aikana testiaskel lähettää suorituskykymittausten tekemisen jälkeen mittaustulokset testiagentille kutsumalla `writeResult`-metodia. Kun testiaskeleen metodien suoritus

päättyy, kontrolli palautuu `JUnitCore`lle, joka tuottaa JUnit-testin suorituksesta tuloksen. Tämä tulos palautetaan edelleen `ClassRunner`-oliolle ja se sisältää muun muassa tiedon testin onnistumisesta sekä testin ajoon kuluneesta ajasta.

Testiaskeleen tulosten kerääminen koostuu seuraavista vaiheista:

1. notifikaation lähettäminen testiagenteille
2. testiaskeleen tulosten kerääminen JUnit-kehykseltä
3. testiaskeleen kokonaistuloksen tuottaminen ja lähettäminen testausjärjestelmän palvelinkomponentille.

Testiaskeleen tulosten kerääminen alkaa, kun testitapausluokka kutsuu testausjärjestelmän ohjaimen rajapintaa. Kutsun seurauksena ohjain julkaisee testiaskeleen tunnisteen sisältävän `TestStepResultRequest`-notifikaation. Kun testiagentti vastaanottaa notifikaation, se kysyy tulokset JUnit-kehykseltä kuvan 4.9 mukaisesti. `JUnitCore`n testiaskeleen suorituksen aikana tuottama tulos palautetaan testiagentille, joka kokoaa saadun tuloksen ja testiaskeleen lähettämien mittaustulosten perusteella testiaskeleen kokonaistuloksen. Lopuksi testiagentti lähettää kokonaistuloksen palvelinkomponentille testitapauksen tuloksen rakentamista varten.

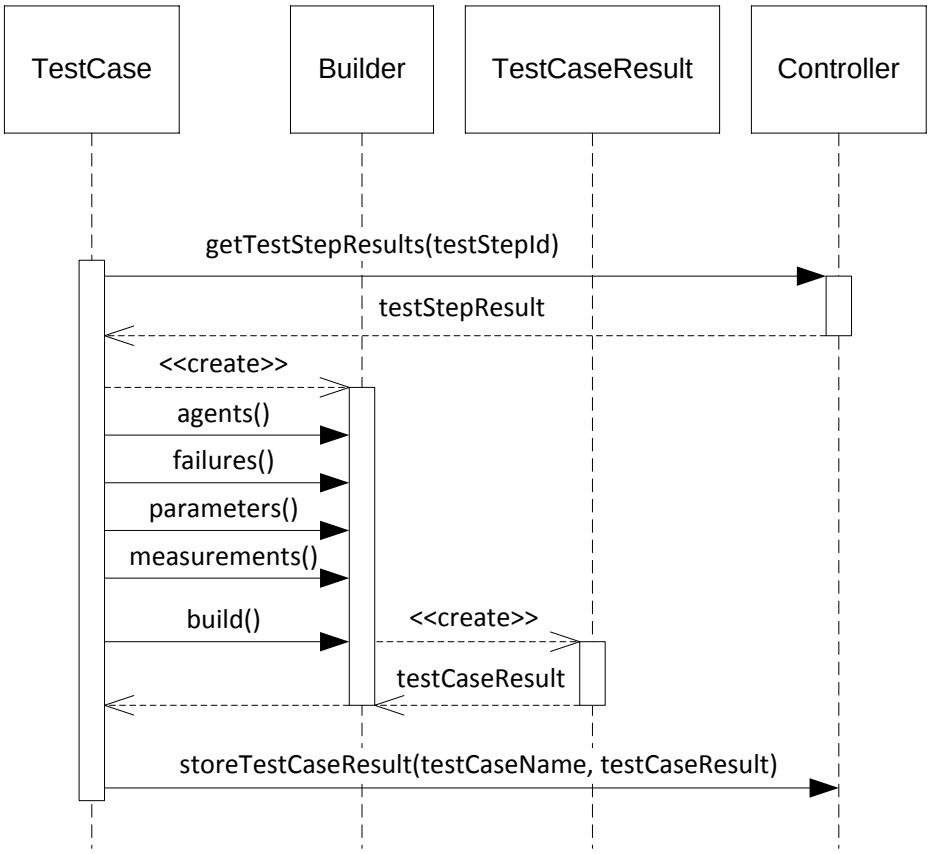
4.5.2 Testituloksen rakentaminen

Testituloksen rakentaminen koostuu seuraavista vaiheista:

1. testiaskelten tulosten kerääminen ohjaimelta testitapausluokalle
2. testin kokonaistuloksen rakentaminen testiaskelten tuloksista
3. testin kokonaistuloksen välittäminen testausjärjestelmän palvelinkomponentille.

Kun testiagentit ovat raportoineet testiaskelten tulokset ohjaimelle, testitapaukset rakentavat testin kokonaistuloksen niiden perusteella. Sekvenssikaavio testin kokonaistuloksen rakentamiseen liittyvästä tyypillisestä kutsujärjestyksestä on esitetty kuvassa 4.10.

Testitapausluokka kysyy testiaskelten tulokset ohjaimelta (`Controller`). Tulosten



Kuva 4.10 *Sekvenssikaavio testituloksen rakentamisesta*

perusteella testitapaus rakentaa testin kokonaistuloksen `Builder`-luokan avulla, jonka metodeilla testitulokseen lisätään tuloksen muodostavat tiedot. Lopuksi tuotettu `TestCaseResult`-tyyppinen testitulos lähetetään ohjaimelle. Ohjaimella sijaitsevien testien kokonaistulosten perusteella voidaan tarvittaessa luoda tulosraportteja.

5. TESTAUSJÄRJESTELMÄN KÄYTTÖ

Tässä luvussa tarkastellaan toteutetun testausjärjestelmän käyttöä ja testien suoritusta. Aluksi kerrotaan testausjärjestelmän käyttöönotosta, asiakassovellusten monistamisesta ja instanssien hallintatyökalusta. Seuraavaksi syvennytään testien käynnistämiseen ja niiden tilan seurantaan. Lopuksi luvussa tarkastellaan testausjärjestelmän tuottamien tulosraporttien rakennetta ja sisältöä.

5.1 Testausjärjestelmän käyttöönotto

Testausjärjestelmän käyttöönotto sisältää palvelin- ja asiakasympäristöjen asennuksen ja käynnistyksen. Testausjärjestelmän palvelinympäristö asennetaan kohdejärjestelmänä toimivaan sovellusalustaan. Palvelinympäristö toimii sovellusalustan näkökulmasta alustaa käyttävänä palveluna, ja sen asennus voidaan suorittaa sovellusalustan asennuksesta erillisenä toimenpiteenä. Testausjärjestelmän palvelinympäristö asennetaan erillisen paketin avulla, ja se voidaan asentaa mihin tahansa olemassa olevaan sovellusalustan ajoympäristöön.

Testausjärjestelmän asiakasympäristön käyttöönotto sisältää asentamisen lisäksi asiakassovellusten monistamisen. Testausjärjestelmän asiakasympäristön tarkoituksena on kehittää hallittavissa oleva kuorma palvelinympäristölle. Kuorma saadaan aikaan monistamalla asiakaskoneille asiakasinstansseja. Monistaminen on toteutettu asiakasympäristön OSGi-ajoympäristönä toimivan Apache Karafin instanssiominaisuudella.

5.1.1 Instanssit

Apache Karaf on avoimen lähdekoodin OSGi-ympäristö, joka koostuu varsinaisen OSGi-toiminnallisuuden sisältävästä OSGi-kehyksestä sekä joukosta lisäominaisuuksia, joita on edellä mainitun instanssi-mekanismin lisäksi muun muassa OSGi-paket-

tien hallintaan käytettävä konsolityökalu sekä SSH-etäyhteys (*Secure Shell*) [1]. Karafin instanssit perustuvat ajoympäristön kopioimiseen. Jokainen yksittäinen instanssi on erillinen, muista instansseista erotettu OSGi-ajoympäristö. Jokaiselle instanssille luodaan oma Javan virtuaalikone (JVM, *Java Virtual Machine*), jossa kunkin instanssin Java-luokat suoritetaan. Luotaville instansseille kopioidaan ainoastaan tarvittavat konfiguraatiotiedostot sekä ajonaikaiset tiedostot sisältävän hakemistot. Karafin instanssien luominen tapahtuu korkealla tasolla seuraavasti:

1. Konfiguroidaan Karafille juuri-instanssi, joka toimii pohjana kopioitaville instansseille.

2. Luodaan Karafin komentorivityökalulla haluttu määrä yksilöivän nimen omaavia instansseja.

3. Karaf kopioi juuri-instanssin konfiguraation ja ajonaikaisen data-hakemiston kullekin instanssille tiedostojärjestelmässä.

4. Kunkin instanssin konfiguraatioihin määritellään yksilöllinen nimi sekä SSH-portti, joiden avulla hallintatoimet voidaan kohdistaa tietylle instanssille.

Luotujen instanssien käynnistys, pysäytys ja muu hallinta tapahtuu paikallisesti Karafin konsolityökalusta tai SSH-etäyhteyden avulla etänä. Karafin komentorivityökalu mahdollistaa uusien instanssien luomisen yksi kerrallaan, mutta se ei tarjoa mahdollisuutta monen instanssin luomiseen tai hallintaan yhden komennon avulla. Tämän vuoksi testausjärjestelmän yhteydessä toteutettiin asiakassovellusten monistamiseen tarkoitettu instanssien hallintatyökalu, jolla useita Karaf-instansseja voidaan hallita yksittäisillä komennoilla.

5.1.2 Instanssien hallintatyökalu

Instanssien hallintatyökalu on Python-ohjelmointikielellä toteutettu käyttöjärjestelmäriippumaton komentorivityökalu, jolla testausjärjestelmän asiakasinstansseja voidaan hallita. Instanssien hallintatyökalulla on tarkoitus helpottaa ja nopeuttaa testausjärjestelmän asiakasympäristöjen pystytystä. Työkalu käyttää Apache Karafin komentorivikonsolia instanssien hallintaan, ja sen avulla voidaan ohjata useita instansseja yksittäisillä komennoilla. Instanssien luomisen, käynnistyksen, pysäytyksen ja poistamisen lisäksi instanssien hallintatyökalu tarjoaa myös muita testausjärjes-

Taulukko 5.1 *Instanssien hallintatyökalun komennot*

Komento	Kuvaus	Parametrit
`create`	Uusien instanssien luonti	*-n* luotavien instanssien lukumäärä *-p* instanssien nimissä käytetty etuliite
`start`	Instanssien käynnistäminen	*-n* käynnistettävien instanssien lukumäärä *-p* käynnistettävien instanssien nimen etuliite
`check`	Käynnistettyjen instanssien tilan tarkistaminen	*-p* tarkistettavien instanssien nimen etuliite
`stop`	Instanssien pysäyttäminen	*-p* pysäytettävien instanssien nimen etuliite
`delete`	Instanssien poistaminen	*-p* poistettavien instanssien nimen etuliite *-a* kaikkien instanssien poistaminen
`log`	Instanssien lokitiedostojen kokoaminen	-
`status`	Instanssilistan näyttäminen	-

telmän käyttöä helpottavia ominaisuuksia. Taulukossa 5.1 esitetään instanssien hallintatyökalun komennot ja käytön kannalta tärkeimmät parametrit.

Uusien instanssien luomiseen käytetään komentoa `create`. Esimerkiksi komennolla `create -n 7 -p agent` työkalu luo seitsemän uutta instanssia, joiden nimet muodostuvat parametrina annetusta etuliitteestä sekä kolminumeroisesta päätteestä seuraavasti: `agent001 - agent007`. Mikäli instanssien hallintatyökalua käytetään monella eri tietokoneella, tulee kullakin koneella luotavien instanssien etuliitteen erota toisistaan yksilöllisten nimien takaamiseksi.

Instanssien käynnistäminen, pysäyttäminen sekä poistaminen tapahtuvat vastaavasti kuin niiden luominen. Asiakastietokoneen suorituskyvystä riippuen kaikki instanssit eivät välttämättä käynnisty oikein ensimmäisellä yrityksellä. Tällaisten tilanteiden varalle työkalu tarjoaa komennon `check`, joka tarkistaa jokaisen käynnistetyn instanssin tilan, ja pyrkii käynnistämään instanssin uudelleen, mikäli vikatilanne ilmenee. Työkalu mahdollistaa myös kaikkien instanssien lokitiedostojen keräämiseen yhteen paikkaan johtuen Karafin instanssien hankalasta hakemistorakenteesta. Ko-

mento `log` kopioi kaikkien instanssien lokitiedostot yhden hakemiston alle ja arkistoi ne lisäksi zip-muotoiseksi paketiksi.

5.2 Testien ajaminen

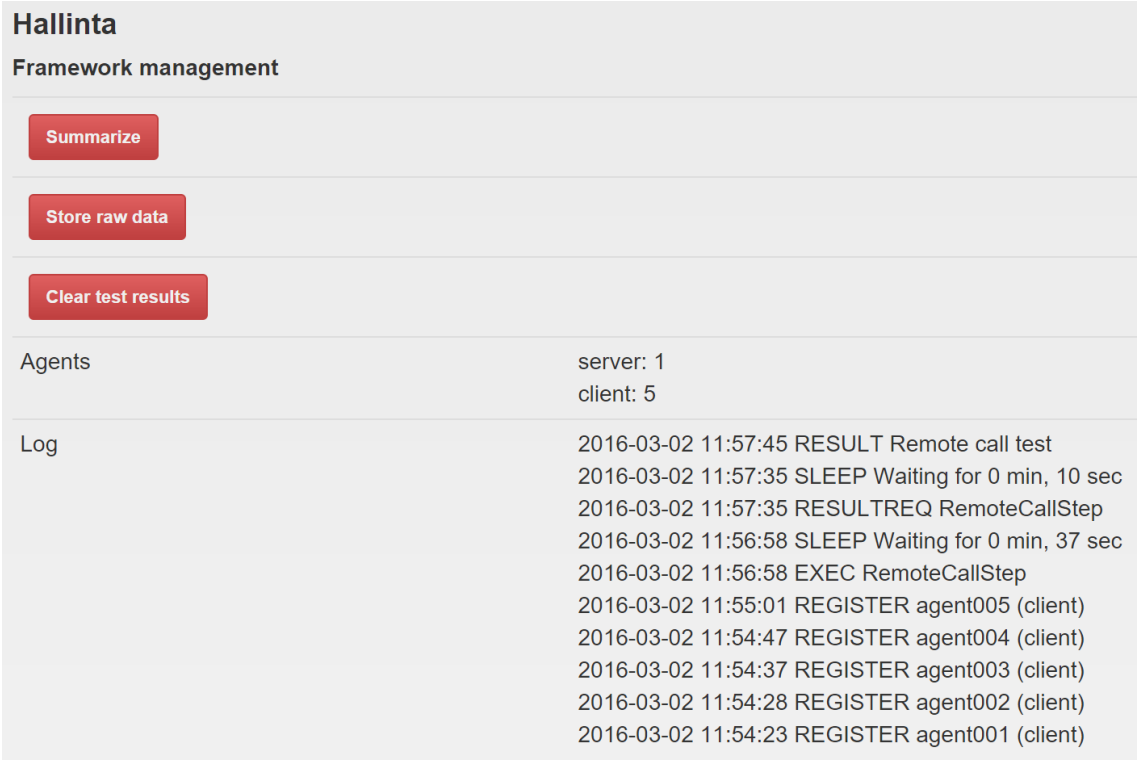
Testausjärjestelmää käytetään sovellusalustan web-pohjaisen järjestelmänhallinnan kautta. Järjestelmänhallintaa käytetään testien viiveiden konfigurointiin, testausjärjestelmän tilan tarkasteluun, testien käynnistämiseen ja niiden tilan seuraamiseen. Ennen testien käynnistämistä tulee tehdä joitain valmistelevia toimenpiteitä.

5.2.1 Ympäristön valmistelu

Testausjärjestelmän käyttö aloitetaan viemällä testausjärjestelmän asiakasympäristön asennuspaketti asiakaskoneille, ja monistamalla ja käynnistämällä haluttu määrä asiakassovelluksia alikohdassa 5.1.2 kuvatulla instanssien hallintatyökalulla. Järjestelmään kirjautuneiden asiakassovellusten määrää voi tarkastella testausjärjestelmän hallinnan kautta. Kuvassa 5.1 on esitetty esimerkkinäkymä testausjärjestelmän hallinnasta. Kuvasta nähdään, että järjestelmään on rekisteröitynyt yksi palvelinympäristön testiagentti (server: 1), sekä viisi asiakasympäristön testiagenttia (client: 5).

Testausjärjestelmän hallinnalla voidaan seurata testien etenemistä juoksevalla lokitulosteella (Log), jossa testausjärjestelmän viestit näytetään käänteisessä järjestyksessä uusin ylimpänä. Lisäksi hallintanäkymästä voidaan tuottaa tulosraportteja (Summarize), tallentaa testien raakadata (Store raw data) ja tyhjentää palvelimen muistissa olevat testitulokset (Clear test results). Testien seurantaan ja tuloksiin liittyviä ominaisuuksia tarkastellaan myöhemmin.

Ennen testien käynnistämistä määritellään lisäksi testeihin liittyvät konfiguraatioarvot käyttäen sovellusalustan konfiguraatiopalvelua. Konfiguroitavat arvot ovat testauspalvelukohtaisia, ja niiden määrittely tapahtuu sovellusalustan järjestelmänhallinnan kautta. Konfiguroitavat arvot voivat olla esimerkiksi arvioita yksittäisten tietokantaoperaatioiden maksimikestoista, joita käytetään testitapauksissa testiaskelten välisten viiveiden määrittelyssä. Arvoja muuttamalla testiaskelten välisiä viiveitä voidaan hallita esimerkiksi palvelimen tai tietokannan suorituskyvyn muuttuessa.



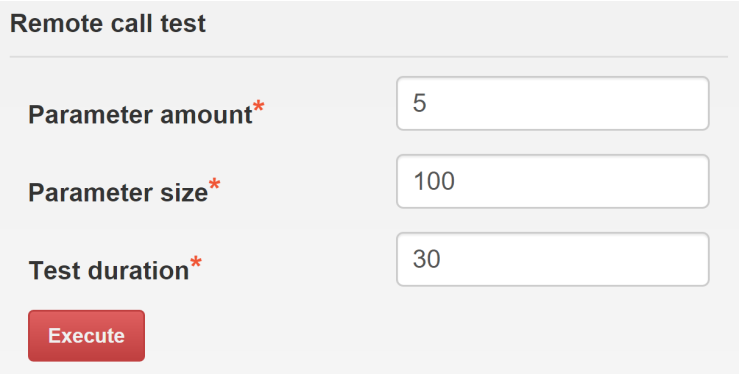
Kuva 5.1 *Näkymä testausjärjestelmän hallinnasta*

5.2.2 Testien käynnistys ja seuranta

Testausjärjestelmän valmistelun jälkeen suorituskykytestien ajaminen voidaan aloittaa. Testit käynnistetään sovellusalustan järjestelmänhallinnan kautta käytettävästä testauspalvelusta. Jokaiselle testille määritellään testauspalveluun mahdolliset testin parametrit sekä painike, josta testi käynnistetään. Esimerkkiote testauspalvelusta löytyvästä testin käynnistyslomakkeesta on esitetty kuvassa 5.2. Parametrien syöttämisen jälkeen kuvassa näkyvä testi käynnistetään Execute-painikkeella, jonka jälkeen testin suoritusta voidaan seurata testausjärjestelmän hallinnasta.

Esimerkki testausjärjestelmän hallinnan lokitulosteesta nähdään kuvassa 5.1 (Log). Juokseva lokituloste näyttää testin suorituksen etenemisen. Lokiviestit koostuvat päivämäärästä, kellonajasta, viestin tyypistä ja itse viestistä. Viestityypit on esitetty taulukossa 5.2.

Tulosteen viisi alinta riviä näyttävät asiakasympäristön testiagenttien rekisteröitymisen testausjärjestelmään. Esimerkkitesti sisältää yhden testiaskeleen *RemoteCallStep* ja notifikaatio sen suorituksen aloittamiseksi lähetettiin ajanhetkellä 11:56:58.



Kuva 5.2 *Testin käynnistys testauspalvelusta*

Taulukko 5.2 *Testien seurantaan käytetyn lokin viestityypit*

Viestin tyyppi	Kuvaus
`EXEC`	Notifikaation lähettäminen testiaskeleen käynnistämiseksi
`REGISTER`	Testiagentin rekisteröityminen
`RESULT`	Testin tuloksen tallennus
`RESULTREQ`	Notifikaation lähettäminen tulosten pyytämiseksi
`SLEEP`	Odotettava aika

Testiaskeleen käynnistyksen jälkeen odotetaan 37 sekuntia, jotta kaikki testin suorittajat saavat testin valmiiksi. Tämän jälkeen testiagenteille lähetetään notifikaatio tulosten raportoimiseksi. Tulosten raportointia odotetaan 10 sekuntia, minkä jälkeen testin tulos tallennetaan palvelimen muistiin.

Kun testin suoritus on päättynyt, testeistä voidaan tuottaa tulosraportit *Summarize*-painikkeella. Painike tuottaa yhden tulosraportin kaikista palvelimen muistissa olevista testituloksista. *Store raw data* -painikkeella kaikista tuloksista saadaan tuotettua raakadataraportti. Palvelimen muistissa olevat testitulokset voi poistaa *Clear test results* -painikkeella.

5.3 Testausjärjestelmän tuottamat tulosraportit

Testausjärjestelmän avulla ajetuista suorituskykytesteistä voidaan generoida valmiita tulosraportteja, joiden avulla sovellusalustan tai sen varaan toteutettujen sovel-

lusten testituloksia voidaan analysoida. Testausjärjestelmä tallentaa ajettujen testien tulokset muistinvaraisesti palvelimelle, ja niiden perusteella luotujen tulosraporttien luonti tapahtuu testausjärjestelmän hallinnan kautta. Testien tuloksista on mahdollista luoda raportit sekä HTML- (*HyperText Markup Language*) että XLSX (*Excel Microsoft Office Open XML Format Spreadsheet File*) -formaateissa. Lisäksi testituloksista on saatavissa myös XML-muotoinen raakadata, joka mahdollistaa tarkemman analyysin tuottamisen valmiiden HTML- ja XLSX-raporttien sijaan. Raporteista ja raakadatasta löytyvät seuraavat tiedot kunkin testiajon osalta:

1. testin nimi
2. testin suorituksen päättymisaika
3. testin onnistuminen (Success, Partial Success, Fail)
4. rekisteröityneiden testiagenttien määrä
5. testiajon parametrit
6. mittaustulokset
7. mahdolliset epäonnistuneet testiaskeleet.

Testiajo on onnistunut (Success), mikäli testiagentit eivät ole raportoineet minkään testiaskeleen epäonnistuneen. Testi on osittain onnistunut (Partial success), mikäli suurin osa tietyn testiaskeleen suorittaneista testiagenteista on raportoinut testin onnistuneen. Esimerkiksi jos tietyn testiaskeleen suorittaa 30 testiagenttia, joista kaksi raportoi askeleen epäonnistuneeksi, testiajo on osittain onnistunut. Testiajo on epäonnistunut, mikäli riittävä määrä yksittäisten testiaskelten ajoista on epäonnistunut. Epäonnistuneeseen testiajoon riittää myös yksi epäonnistunut testiaskel, mikäli kyseinen testiaskel suoritetaan vain yhdellä testiagentilla (esim. palvelimella).

Testiraporteista ilmenee rekisteröityneiden testiagenttien määrä kunkin testiagenttiluokan mukaan (esim. palvelin, asiakkaat). Raportissa näkyvät testiagenttien lukumäärät viittaavat palvelinympäristöön rekisteröityneiden agenttien määrään, eivätkä siis välttämättä vastaa täysin testiä suorittaneiden agenttien määrää. Raporttien varsinaiset mittaustulokset esitetään tulosraporteissa sen mukaan, miten ne on määritelty testitapauksissa (alikohta 4.4.2). Jokaisesta mitatusta suureesta esitetään

Test summary (Mon Feb 15 08:35:04 EET 2016)

Remote call test SUCCESS

Timestamp: Mon Feb 15 08:28:44 EET 2016

Agents

Type	Count
client	5
server	1

Parameters

Name	Value
Parameter count	5
Parameter size (byte)	100
Test duration (sec)	30

Measurements

Measurement	n	Min	Max	Mean	Median
Remote call processing delay of agent001 (ms)	1250	1.0	1096.0	23.97	14.0
Remote call processing delay of agent002 (ms)	1284	1.0	678.0	23.36	16.0
Remote call processing delay of agent003 (ms)	1276	1.0	695.0	23.51	12.84
Remote call processing delay of agent004 (ms)	1237	1.0	634.0	24.25	11.64
Remote call processing delay of agent005 (ms)	1263	1.0	664.0	23.75	16.0
All remote call processing delays (ms)	6310	1.0	1096.0	23.76	16.0

Kuva 5.3 *HTML-muotoinen tulosraportti*

mittausten lukumäärän lisäksi seuraavat tunnusluvut: pienin arvo, suurin arvo, keskiarvo ja mediaani. Mikäli yksi tai useampi testiaskel on epäonnistunut, esitetään kyseisten testiaskelten nimi sekä askelen suorittaneen testiagentin nimi omassa listassaan kunkin testiajoon liittyvän raportin lopussa. Esimerkki onnistuneesta testiajosta tuotetusta HTML-muotoisesta raportista on esitetty kuvassa 5.3.

Esimerkkiraportin Agents-kohdasta havaitaan, että testausjärjestelmän palvelinkomponentille on rekisteröitynyt viisi asiakasympäristön testiagenttia ("client") sekä yksi palvelinympäristön testiagentti ("server"). Testiajon parametrit kuvataan taulukossa kohdassa Parameters. Measurements-osio sisältää varsinaiset mittaustulokset, jotka on määritelty testitapauksen yhteydessä. Esimerkkitestissä etäkutsujen viiveet on mitattu jokaiselle asiakasympäristön testiagentille erikseen. Näistä mittaustuloksista esitetään edellä mainitut tunnusluvut.

Valmiiden HTML- ja XLSX-raporttien sisältäessä valmiiksi lasketut tunnusluvut XML-muotoinen raakadataraportti sisältää kunkin mittauksen jokaisen mitatun arvon. Näistä raakatuloksista voidaan tarvittaessa tuottaa laajojakin testiraportteja. XML-raporttien tarkoitus on tarjota mahdollisuudet syvällisempään testitulosten analysointiin, kuin mitä valmiit raportit mahdollistavat.

6. TULOSTEN TARKASTELU JA ARVIOINTI

Tässä diplomityössä kehitetään kohdejärjestelmänä toimivalle sovellusalustalle yleiskäyttöinen testausjärjestelmä, jolla voidaan testata sovellusalustan tarjoamien palveluiden suorituskykyä. Lisäksi kehitettävän testausjärjestelmän tulee tarjota mahdollisuudet sovellusalustan varaan rakennettujen sovellusten suorituskyvyn testaamiseen. Tässä luvussa arvioidaan, kuinka hyvin toteutettu testausjärjestelmä vastaa sille asetettuihin vaatimuksiin ja tavoitteisiin, sekä tarkastellaan testausjärjestelmän käytöstä saatuja kokemuksia. Lopuksi luvussa pohditaan testausjärjestelmän jatkokehitykseen liittyviä mahdollisuuksia.

6.1 Testausjärjestelmän arviointi

Tässä kohdassa toteutettua testausjärjestelmää arvioidaan vertaamalla sitä sen tavoitteisiin (alikohta 3.3.1) ja sille asetettuihin vaatimuksiin (alikohta 3.3.2). Sovellusalustan suorituskykyä ei ole testattu järjestelmällisesti ennen testausjärjestelmän kehitystä, joten saaduille tuloksille ei ole vertailua mahdollistavaa aiempaa pohjaa.

Toteutettu testausjärjestelmä täyttää sille asetetut tavoitteet, joiden mukaan sen tulisi tarjota mahdollisuus sovellusalustan palvelinympäristön suorituskyvyn testaamiseen. Testausjärjestelmän avulla sovellusalustaa voidaan testata tavoitteiden mukaisessa tilanteessa, jossa palvelimelle tehdään runsaasti rinnakkaisia palvelupyyntöjä. Tämän mahdollistavat monistettavat asiakasinstanssit, joiden avulla riittävä rinnakkainen kuorma voidaan luoda hallitusti ja samanaikaisesti. Järjestelmän avulla asiakassovelluksia voidaan monistaa suuriakin määriä, mikäli testiympäristön kapasiteetti tämän mahdollistaa.

Vaatimus testausjärjestelmän siirrettävyydestä on osoitettu täyttyneeksi käyttämällä testausjärjestelmää eri käyttöympäristöissä. Testausjärjestelmän palvelinympäristö asennetaan kohdejärjestelmänä toimivan sovellusalustan yhteyteen, ja se on siirrettävissä ja asennettavissa erillisenä pakettina. Koska testausjärjestelmän palveli-

nympäristö on riippuvainen sovellusalustan asennuksesta, tämä vaatimus kohdistuu pääasiassa testausjärjestelmän asiakasympäristön siirrettävyyteen. Asiakasympäristö toimitetaan yhdessä noin 70 megatavun kokoisessa paketissa, jonka asennus vaatii vain paketin purkamisen. Testausjärjestelmän ajaminen vaatii, että ympäristöön on asennettu Java JDK (*Java Development Kit*) sekä Python-ajoympäristö. Testausjärjestelmän asiakasympäristö toimii Windows- ja Linux-käyttöjärjestelmillä.

Vaatimus testausjärjestelmän ajoympäristön vastaavuudesta lopullisen tuotantoympäristön konfiguraation kanssa sovelluspalvelimen ja oheisohjelmien osalta on myös täytetty. Sovelluspalvelimen ajoympäristö perustuu OSGiin, joten vaatimuksen täyttymiseksi myös testausjärjestelmän tulee olla OSGi-yhteensopiva. Testausjärjestelmä on kehitetty OSGi-yhteensopivaksi, ja testausjärjestelmää voidaan ajaa täysin sovelluspalvelimen tuotantoympäristöä vastaavissa konfiguraatioissa. Testausjärjestelmää voidaan siis käyttää myös sovellusalustan tai sen varaan toteutettujen sovellusten hyväksymistestauksen yhteydessä.

Testausjärjestelmän asiakasympäristön asennuksen helppouden ja instanssien hallintatyökalun avulla asiakassovelluksia voidaan monistaa useille asiakaskoneille. Kuten edellä kuvattiin, asiakasympäristön asentaminen sisältää vain asennuspaketin purkamisen, jonka jälkeen asiakasympäristö on käyttövalmis, ja asiakassovelluksien monistaminen voidaan aloittaa instanssien hallintatyökalulla. Saatujen kokemusten perusteella asiakasympäristön asentaminen sekä instanssien luominen ja käynnistäminen vievät instanssien lukumäärästä riippuen joitain minuutteja olettaen, että asiakaskone täyttää asiakasympäristön esivaatimukset. Vaatimus voidaan siis katsoa täyttyneeksi.

Vaatimus asiakassovellusten ohjaamisesta komentorivin kautta on täyttynyt. Asiakassovelluksia voidaan ohjata testausjärjestelmän mukana tulevalla instanssien hallintatyökalulla. Työkalu tarjoaa riittävät toiminnot sovellusten ohjaamiseen, ja sen avulla asiakassovelluksia voidaan luoda, käynnistää, sulkea ja tuhota. Instanssien hallintatyökalu on toteutettu Pythonilla ja sitä käytetään komentorivin kautta.

Testausjärjestelmä vastaa kaikkiin sille asetettuihin vaatimuksiin. Järjestelmälle asetettiin kuitenkin myös täydentäviä vaatimuksia, joiden toteutumista tarkastellaan seuraavaksi.

Testausjärjestelmä mahdollistaa sen laajentamisen uusilla testitapauksilla. Uuden testitapauksen toteuttaminen vaatii testitapauksen ja siihen liittyvien testiaskelten

toteuttamisen sekä testitapauksen lisäämisen testauspalveluun. Testausjärjestelmä tarjoaa uusille testitapauksille ja testiaskelille kantaluokat, jotka tarjoavat yleisiä luokkien toteutuksessa vaadittavia metodeita. Uusia testikokonaisuuksia varten voidaan tarvittaessa toteuttaa uusi testauspalvelu. Vaatimus voidaan siis katsoa täytetyksi.

Testausjärjestelmä ei aseta rajoitteita testattaville kohteille. Toteutettavien testitapausten ja -askelten toiminnallisuutta ei rajoiteta millään tavalla, joten minkä tahansa sovellusalustan kautta tarjottavan tai sen varaan toteutetun, saatavilla olevan palvelun suorituskykyä voidaan testata. Vaatimus sovellusalustan varaan toteutettujen sovellusten testaamisesta sovelluskohtaisilla testitapauksilla voidaan siis katsoa täyttyneen.

Täydentävien vaatimusten lisäksi testausjärjestelmän toteutuksen kannalta oli toivottavaa, että se hyödyntäisi sovellusalustan tarjoamia palveluita, jotta testausjärjestelmän toteutuksen yhteydessä vältyttäisiin uusien mekanismien toteuttamiselta. Toteutetun testausjärjestelmän kommunikoinnin ydintoiminnot on toteutettu käyttäen sovellusalustan tarjoamaa notifikaatiopalvelua sekä etäkutsukehystä. Lisäksi testien ohjaamiseen käytetään sovellusalustan järjestelmänhallintaa, ja testiaskelten välisten viiveiden konfigurointiin voidaan käyttää konfiguraatiopalvelua.

6.2 Käyttökokemuksia

Työssä kuvattua suorituskykytestausjärjestelmää on käytetty kohdejärjestelmänä toimivan sovellusalustan suorituskykytestaukseen. Testausjärjestelmällä on toteutettu laajoja suorituskykytestejä sovellusalustan eri osien suorituskyvyn mittaamiseen. Tässä kohdassa tarkastellaan tähän mennessä saatuja kokemuksia järjestelmän käytöstä ja testien toteuttamisesta.

Testien toteuttaminen

Järjestelmä mahdollistaa monipuolisten ja laajojen suorituskykytestien toteuttamisen. Testausjärjestelmä on yleiskäyttöinen, eikä se aseta juurikaan rajoitteita testitapausten suunnittelulle. Näin testausjärjestelmällä voidaan kehittää hyvin erilaisia testejä eritasoisten kokonaisuuksien testaamiseen. Testi voi olla esimerkiksi hyvin yksinkertainen, yhden testiaskeleen sisältävä testi, joka testaa yhden palvelinympäristön metodin suorituskykyä. Toisaalta järjestelmä mahdollistaa myös laajojen,

useista testiaskeleista ja vaiheista koostuvien kokonaisuuksien toteuttamisen.

Järjestelmän yleiskäyttöisyys aiheuttaa kuitenkin testitapausten kehittämisen raskauden. Yhden ei-triviaalin testitapauksen kehittäminen vaatii kohtalaisen paljon ohjelmointia, ja osa ohjelmakoodista on ns. *boilerplate*-koodia, joka toistuu samana tai hyvin samankaltaisena testitapausten välillä. Keskimäärin yhden testin toteuttamiseen kuluu sen laajuudesta riippuen 0,5–2 työpäivää. Yksittäisen testin toteutus vaatii uuden tiedoston luomisen testitapaukselle sekä jokaiselle testiaskeleelle. Lisäksi muutoksia tarvitaan useisiin olemassa oleviin tiedostoihin, joten yhden testiaskeleen sisältävän testin toteutus vaatii toimenpiteitä vähintään kymmeneen tiedostoon.

JUnit on havaittu toimivaksi ajoympäristöksi testiaskelille. JUnitin mekanismit mahdollistavat monipuolisten testiaskelten toteuttamisen, eikä testiaskelten toteutuksessa ole havaittu mitään rajoitteita, jonka vuoksi yksikkötestaukseen tarkoitettu JUnit ei soveltuisi suorituskykytestien kehitykseen.

Viiveiden määrittely

Testitapausten aikana suoritettavien testiaskelten väliin tulee määritellä viive, jonka aikana kaikki testiaskeleen suorittavat testiagentit ehtivät suorittaa testin varmasti loppuun. Tämän viiveen määrittely perustuu siihen, että testausjärjestelmän asiakas- ja palvelinympäristö eivät kommunikoi testiaskelten suoritusten välillä millään lailla, jotta kommunikointi ei kuormita testattavaa järjestelmää. Sopivien viiveiden määrittely testiaskelten väliin on kuitenkin hankalaa, ja usein testiaskelten suoritusaika riippuu testin parametreista ja testiagenttien määrästä.

Viiveiden määrittelyn haasteista voidaan ottaa esimerkkinä testi, jossa testiagentit tekevät rinnakkain kirjoitusoperaatioita tietokantaan siten, että eräänä testin parametrina on kirjoitettavien rivien lukumäärä. Testin kestoa ei ole siis etukäteen määritelty, kuten aiemmin työssä käytetyssä esimerkkitestissä. Tietokantaoperaation sisältävän testiaskeleen jälkeinen viive tulee siis määritellä testiaskeleen suoritusajan mukaan, joka riippuu vahvasti testiagenttien lukumäärästä sekä kirjoitettavien rivien lukumäärästä. Tällaisen testiaskeleen suoritusaikaan vaikuttaa myös testiympäristön rakenteen, verkon viiveen sekä palvelimen suorituskyvyn lisäksi tietokannan suorituskyky ja rinnakkaisten yhteyksien maksimimäärä. Mahdollisten ympäristön muutosten mukanaan tuomat testiaskelten suoritusaikojen vaihtelut voidaan kui-

tenkin huomioida käyttämällä viiveissä sovellusalustan konfiguraatiopalvelun kautta muokattavia arvoja. Näin testien ajon yhteydessä viiveitä voidaan korjata muuttamalla arvoja konfiguraatiopalvelun kautta.

Testiaskelten suoritusaika riippuu siis usein monesta eri tekijästä, jotka mahdollisesti vaikuttavat huomattavasti yksittäisten testiaskelten suoritusaikaan. Mikäli viive määritellään liian lyhyeksi, seuraava testiaskel käynnistetään mahdollisesti liian aikaisin, jolloin päällekkäin suoritetut askeleet voivat jättää testattavan palvelun alkuperäisestä poikkeavaan tilaan. Tällainen tilanne voi ilmetä, mikäli tietokannan tyhjentävä testiaskel käynnistetään, kun edellinen, tietokantaan kirjoittava testiaskel on yhä kesken. Liian pitkäksi määritelty viive taas kasvattaa turhaan testien ajoon tarvittavaa aikaa, joka usein kertautuu parametrien arvojen kasvaessa.

Sekvenssien käyttö

Useista testitapauksien ajoista koostuvien testisekvenssien toteuttaminen XML-tiedostojen avulla on yksinkertaista. Sekvenssien käyttö vähentää huomattavasti testien suoritukseen vaadittavaa aikaa testaajien osalta. Testausjärjestelmän tarjoaman sekvenssien generointityökalun käyttö helpottaa sekvenssien tekemistä entisestään, ja sen käytön myötä sekvenssien XML-skeemaa ei tarvitse myöskään hallita, vaan työkalu generoi XML-tiedoston testien nimien ja parametrien perusteella. Sekvenssit mahdollistavat myös työajan ulkopuolisen ajan käyttämisen testaukseen, jolloin esimerkiksi arkiöiden ja viikonloppujen hiljainen aika voidaan käyttää hyödyksi suorittamalla pitkiä testisekvenssejä.

Sekvenssien käyttöä kuitenkin hankaloittavat niiden vaatimat `sleep`-elementit, joilla määritellään testitapausten välissä odotettava aika. Käytännössä elementtiin määritettävä aika on testitapauksen suoritukseen kuluva aika, joka pitää laskea toteutetun testitapausluokan yksittäisten testiaskelten välisten viiveiden summana. Nämä viiveet riippuvat usein testitapaukselle annetuista parametreista, jolloin viive tulee laskea testitapauksen jokaiselle parametriyhdistelmälle erikseen. Viiveiden laskeminen voidaan kuitenkin jättää testausjärjestelmän tarjoaman sekvenssien generointityökalun vastuulle, johon voidaan määritellä testitapauskohtaisesti tapauksen kesto parametrimuuttujien avulla. Viivefunktio tulee näin määritellä jokaiselle toteutetulle testitapaukselle. Tämä hankaloittaa testitapausten ylläpitoa, koska mikäli testitapauksen sisäiset viiveet muuttuvat, tulee myös sekvenssien generointityökaluun määritellyn viivefunktion arvoa muuttaa vastaavasti.

Testausjärjestelmän käyttö

Asiakassovellusten monistaminen eri koneille onnistuu testausjärjestelmän asiakasympäristön tarjoaman komentorivikäyttöliittymän avulla tehokkaasti. Testausjärjestelmä ei kuitenkaan tällä hetkellä tarjoa mahdollisuutta muuttaa asiakassovellusten määrää ajonaikaisesti. Tämä rajoittaa joidenkin testityyppien soveltamista. Esimerkiksi ramp-up-testien toteuttaminen ei järjestelmällä ole mahdollista. Lisäksi ominaisuuden puuttuminen aiheuttaa sen, että mikäli testien suorittamiseen käytetään järjestelmän tarjoamia testisekvenssejä, jokainen sekvenssiin määritelty testi ajetaan samalla, ennen sekvenssin aloitusta, määritellyllä asiakasmäärällä. Tämä rajoittaa sekvenssien käyttömahdollisuuksia.

Testausjärjestelmän tuottamista HTML- ja XLSX-testiraporteista saadaan selkeät tiedot testien onnistumisesta ja niiden mittaustuloksista. Raporttien esittämät tunnusluvut tarjoavat myös tärkeää tietoa mittaustuloksista. Raporttien ulkonäkö on selkeä, ja testiagenttien määrät, testin parametrit sekä mittaustulokset ovat selkeästi luettavissa. XLSX-muotoisten raporttien avulla mittaustuloksista voidaan tarvittaessa luoda taulukkolaskentaohjelmassa kaavioita ja analysoida tuloksia tarkemmin. Testausjärjestelmän tuottama XML-muotoinen raakadataraportti mahdollistaa laajemman analyysin tekemisen niiden sisältämien yksittäisten mittaustulosten vuoksi.

Testien etenemistä voi seurata testausjärjestelmän hallinnan kautta. Järjestelmään rekisteröityneiden testiagenttien lukumäärä ja juokseva lokituloste tarjoavat riittävän kuvan testausjärjestelmän tilasta. Hallinnan kautta ei kuitenkaan ole suoraan nähtävissä esimerkiksi kyseisellä ajanhetkellä ajossa oleva testi, vaan suoritettava testi pitää päätellä lokitulosteista käynnistettyjen testiaskelten perusteella. Mikäli testeissä ilmenee virheitä, niiden tarkastelu onnistuu Javan lokikirjastojen tuottamien lokitiedostojen perusteella, joita on palvelinympäristölle yksi ja kutakin asiakasinstanssia kohden yksi. Testien kehityksen apuna voidaan myös käyttää debugtulosteita, joiden avulla lokitiedostoista voidaan havaita mahdollisia virheitä testeissä.

Testien käynnistäminen ja parametrisointi testauspalvelun kautta on suoraviivaista. Testauspalvelun määrittelyn yhteydessä testitapauksille annettavien parametrien validointi voidaan suorittaa niiden syöttämisen yhteydessä, jolloin mahdollisesti vää-

rän tyyppiset arvot eivät käynnistä testitapauksen suoritusta.

6.3 Jatkokehitysehdotukset

Diplomityön kirjoittamishetkellä testausjärjestelmä tarjoaa tässä työssä kuvatut ominaisuudet. Ne tarjoavat riittävät ominaisuudet suorituskykytestauksen tarpeisiin ja niiden avulla voidaan toteuttaa kattavat testit sovellusalustalle tai sen varaan toteutetuille sovelluksille. Testausjärjestelmässä on kuitenkin vielä monia osia, joita voi jatkossa kehittää. Tässä kohdassa on esitelty ajatuksia testausjärjestelmän jatkokehityksestä, joilla testausjärjestelmästä saavutettavaa hyötyä voitaisiin edelleen kasvattaa.

Suorituskykytestausta voidaan hyödyntää regressiotestauksen yhteydessä selvittämään tehtyjen muutosten vaikutusta järjestelmän suorituskykyyn. Testausjärjestelmän tuottamia tulosraportteja voitaisiin hyödyntää tehokkaammin regressiotestauksen yhteydessä, mikäli ne esittäisivät haluttaessa mitattujen tulosten lisäksi vertailun edellisiin tuloksiin. Ominaisuus olisi toteutettavissa järjestelmään siten, että testausjärjestelmän palvelinympäristössä säilytettäisiin kunkin testin viimeisimpiä tuloksia. Edelleen pidemmälle vietynä järjestelmä voisi mahdollistaa tulosten historiatiedon esittämisen, jolloin ajettujen testien tuloksia voisi verrata mihin tahansa aiemmin saaduista tuloksista.

Testausjärjestelmän tulosten analysointia voisi automatisoida. Järjestelmä voisi tarvittaessa tuottaa testin kokonaistulosten perusteella automaattisia kaavioita, joissa esimerkiksi mittaustulokset eri parametrien arvoilla esitettäisiin pylväs- tai viivakaavioina. Näin nykyisten mittausarvojen ja niiden perusteella laskettujen tunnuslukujen lisäksi suorituskyvystä saataisiin visuaalista palautetta.

Testitapausten sisältämien testiaskelten välisten viiveiden epäonnistunut määrittely voi aiheuttaa epätoivottuja tilanteita testien ajon yhteydessä. Testiaskelten todelliset suoritusajat riippuvat usein monesta tekijästä, jolloin niiden määrittely eri muuttujien suhteen täsmällisesti voi osoittautua hyvin vaikeaksi. Testitapausten toteutus yksinkertaistuisi, mikäli viiveitä ei olisi tarvetta määritellä käsin. Testausjärjestelmän toiminta kuitenkin perustuu siihen, että järjestelmän eri osat eivät kommunikoi toistensa kanssa kesken testiaskelten suorituksen, jotta kommunikoinnin aiheuttama kuorma ei häiritse järjestelmän toimintaa testiaskelten suorituksen aikana. Käsin määriteltävien viiveiden tarpeen poistamiseksi testiagenteilla tulisi olla jokin tapa

ilmoittaa testiaskeleen suorituksen päättymisestä palvelinympäristölle käyttämättä sovellusalustan tarjoamia keinoja. Tämän mahdollistamiseksi testausjärjestelmään voisi kehittää erillisen kommunikointimekanismin, joka ei perustu sovellusalustan tarjoamiin palveluihin, eikä niiden käyttämiin teknologioihin.

Testausjärjestelmän hallinta voisi mahdollistaa testin ajon keskeytyksen. Keskeyttäminen on tarpeellista esimerkiksi tilanteessa, jossa testi on käynnistetty väärillä parametrien arvoilla. Itse testitapauksen keskeyttäminen on triviaalia, ja testiaskelten ohjainkomponentin JUnit-kehys tarjoaa mekanismin testiaskelten suorituksen keskeyttämiseksi. Tieto testiaskeleen keskeytyksestä voitaisiin lähettää testiagenteille erillisen notifikaation avulla, joka sisältää keskeytettävän testiaskeleen tunnisteen.

Eräs suorituskyvyn yhteydessä mitattavista asioista on järjestelmän käyttöaste. Toteutettu testausjärjestelmä ei tällä hetkellä tarjoa tietoa palvelimen käyttöasteesta suorituskykytestien aikana. Tieto käyttöasteesta eri suorituskykytestien aikana olisi hyödyllinen, ja testausjärjestelmää voisi kehittää siten, että se tarjoaisi rajapinnan, jolla palvelimen käyttöasteesta saataisiin tietoa testiaskelten suorituksen aikana. Tämän rajapinnan kautta saatuja tietoja voisi lisätä mittaustuloksina testin kokonaistuloksiin, jolloin ne voitaisiin esittää tulosraporteissa. Järjestelmän monitorointiin tarjoavat mahdollisuuksia Javan oman `java.lang.management`-paketin lisäksi joukko erillisiä kirjastoja, kuten Spf4j (*Simple performance framework for Java* [10]).

Asiakasinstanssien hallinta tapahtuu komentoriviltä käytettävän instanssien hallintatyökalun avulla, eikä instanssien käynnistys tai pysäyttäminen ole mahdollista ajonaikaisesti testausjärjestelmän hallinnan kautta. Instanssien ajonaikainen hallinta mahdollistaisi monipuolisempien sekvenssien toteuttamisen, joissa eri testiajoille voitaisiin määritellä testin suorittavien instanssien määrä. Tämän lisäksi ominaisuuden myötä järjestelmä mahdollistaisi esim. ramp-up-testien suorituksen. Käytännössä instanssien hallinta ohjelmallisesti palvelinympäristön kautta onnistuisi Karafajoympäristön tarjoamien JMX- (*Java Management Extensions*) tai SSH-yhteyksien avulla.

Kattavien suorituskykytestien suorittaminen on pitkäkestoinen operaatio. Esimerkiksi tiedon pysyvään tallentamiseen liittyvien testien ajaminen suurilla parametrien arvoilla voi viedä useita päiviä. *Littlen laki* on jonoteorian laki, jota käytetään prosessien läpimenoajan määrittämiseen [19]. Hyödyntämällä Littlen lakia parametrisoitavien testien tuloksia voitaisiin arvioida myös sellaisilla parametrien arvoilla, joilla testejä ei ole ajettu. Näin esimerkiksi järjestelmän vasteaikoja olisi mahdol-

lista arvioida suurilla parametrien arvoilla ja mahdollisia automaattisesti luotavien kaavioiden arvoja voitaisiin ekstrapoloida suuremmille arvoille [21].

7. YHTEENVETO

Tässä diplomityössä selvittiin, miten sovellusalustan suorituskykytestit ja niiden raportointi voidaan automatisoida, jotta saadaan hyödyllisiä ja vertailukelpoisia tuloksia järjestelmän suorituskyvystä. Kohdejärjestelmänä toimiva sovellusalusta on Java-pohjainen ohjelmisto- ja teknologia-alusta, jonka on tarkoitus toimia yleiskäyttöisenä johtamisjärjestelmäalustana sen varaan kehitettäville toimialakohtaisille loppukäyttäjäsovelluksille. Sovellusalustan ja sen sovellusten palvelinympäristöjen suorituskyvyn testaamiseen kehitettiin suorituskykytestausjärjestelmä, jonka rakennetta ja toimintaa esiteltiin ja arvioitiin tässä työssä.

Työn alussa tarkasteltiin suorituskykytestausta, sen mittaamista ja eri muotoja. Lisäksi tarkasteltiin testauksen automatisoinnin hyötyjä ja ongelmia sekä tehtiin katsaus testauksen apuna käytettäviin työkaluihin. Tämän jälkeen esiteltiin kohdejärjestelmänä toimiva sovellusalusta ja kuvattiin toteutettavan testausjärjestelmän tavoitteet ja vaatimukset. Seuraavaksi esiteltiin toteutetun testausjärjestelmän rakenne ja toiminta, tarkasteltiin järjestelmälle toteutettavien suorituskykytestien määrittelyä ja esiteltiin testausjärjestelmän käyttöä. Lopuksi testausjärjestelmän toteutuksen onnistumista arvioitiin sen käytöstä saatujen käyttökokemusten perusteella sekä vertaamalla sitä sille asetettuihin tavoitteisiin ja vaatimuksiin. Lisäksi esiteltiin ehdotuksia järjestelmän jatkokehittämiseksi.

Ennen testausjärjestelmän toteutusta tehtiin kartoitus markkinoilla oleviin valmiisiin testaustyökaluihin, joita suorituskykytesteissä olisi voitu hyödyntää. Markkinoilla olevat avoimen lähdekoodin ratkaisut eivät kuitenkaan tarjonneet riittäviä ominaisuuksia, minkä vuoksi testausjärjestelmä kehitettiin itse.

Toteutetun testausjärjestelmän avulla asiakassovellukset voivat luoda hallitusti kuormaa sovellusalustan palvelinympäristöön usealta eri asiakaskoneelta. Testausjärjestelmä on yleiskäyttöinen, ja sen avulla voidaan toteuttaa monipuolisia suorituskykytestejä. Testitapaukset määritellään Java-luokkina, ja ne koostuvat testiluokista, joiden ajamiseen käytetään JUnit-yksikkötestauskehystä.

Testausjärjestelmän todettiin täyttävän sille asetetut tavoitteet ja vaatimukset. Järjestelmän avulla on toteutettu kattavat suorituskykytestit sovellusalustan palveluille, ja lisäksi sitä voidaan käyttää alustan varaan kehitettyjen sovellusten testaamiseen. Testausjärjestelmä tarjoaa riittävät ominaisuudet suorituskykytestien toteuttamiseen ja tulosten tarkasteluun, mutta testausjärjestelmää voidaan edelleen kehittää monella eri tavalla.

Jatkossa toteutettua testausjärjestelmää voidaan käyttää sovellusalustan varaan kehitettyjen uusien sovellusten testaamiseen. Lisäksi sovellusalustan palveluita testaavia suorituskykytestejä voidaan edelleen laajentaa kattamaan suurempi osa järjestelmän palveluista.

LÄHTEET

[1] Apache Karaf. The Apache Software Foundation. Viitattu 08.02.2016. Saatavissa: http://karaf.apache.org/

[2] JMeter. The Apache Software Foundation. Viitattu 28.02.2016. Saatavissa: http://jmeter.apache.org/

[3] JUnit. Viitattu 28.02.2016. Saatavissa: http://junit.org/

[4] JUnit on OSGI. Viitattu 06.02.2016. Saatavissa: http://www.knopflerfish.org/releases/5.2.0/docs/bundledoc/index.html?docpage=junit/index.html

[5] Junit4OSGi. Viitattu 06.02.2016. Saatavissa: http://felix.apache.org/documentation/subprojects/apache-felix-ipojo/apache-felix-ipojo-tools/junit4osgi.html

[6] Manual testing. Techopedia Inc. Viitattu 28.02.2016. Saatavissa: https://www.techopedia.com/definition/29843/manual-testing

[7] OSGi Alliance - The Dynamic Module System for Java. Viitattu 18.01.2016. Saatavissa: https://www.osgi.org/

[8] OSGi Testrunner. Viitattu 06.02.2016. Saatavissa: http://www.everit.org/osgi-testrunner/

[9] Pax Exam - OPS4J - Pax Exam 4.x. Viitattu 06.02.2016. Saatavissa: https://ops4j1.jira.com/wiki/display/PAXEXAM4

[10] Spf4j - Simple performance framework for Java. Viitattu 29.02.2016. Saatavissa: http://zolyfarkas.github.io/spf4j/

[11] TestNG. Viitattu 28.02.2016. Saatavissa: http://testng.org/

[12] The Top Five Challenges of Building Software Platforms in the Agile World. C4Media Inc. Viitattu 28.02.2016. Saatavissa: http://www.infoq.com/articles/challenges-building-sw-platforms-with-agile

[13] Using JConsole. Oracle. Viitattu 28.02.2016. Saatavissa: https://docs.oracle.com/javase/8/docs/technotes/guides/management/jconsole.html

[14] When to automate your testing (and when not to). Oracle. Viitattu 28.02.2016. Saatavissa: http://www.oracle.com/technetwork/topics/qa-testing/whatsnew/when-to-automate-testing-1-130330.pdf

[15] L. Crispin and J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, ser. Addison-Wesley Signature Series (Cohn). Pearson Education, 2008.

[16] M. Fewster and D. Graham, *Software Test Automation: Effective use of test execution tools*. Great Britain: ACM Press, 1999, 562 p.

[17] I. Haikala and T. Mikkonen, *Ohjelmistotuotannon käytännöt*. Helsinki: Talentum Media Oy, 2011, 242 s.

[18] M. Katara, M. Vuori, and A. Jääskeläinen. Ohjelmistojen testaus. Tampere. Viitattu 29.12.2015. Saatavissa: http://www.cs.tut.fi/~testaus/s2015/luennot/TIE-21204_2015.pdf

[19] J. D. C. Little, "A proof of the queueing formula L = λW ," *Operations Research*, vol. 9, pp. 383–387, 1961.

[20] M. Maccaux. Approaches to Performance Testing. Oracle. Viitattu 28.02.2016. Saatavissa: http://www.oracle.com/au/products/database/performance-testing-095962.html

[21] R. Mansharamani, A. Khanapurkar, B. Mathew, and R. Subramanyan, "Performance testing: Far from steady state," in *Computer Software and Applications Conference Workshops (COMPSACW), 2010 IEEE 34th Annual*, July 2010, pp. 341–346.

[22] J. Meier, *Performance Testing Guidance for Web Applications: Patterns & Practices*, ser. ITPro collection. Microsoft Press, 2007, 256 p. Saatavissa: https://msdn.microsoft.com/en-us/library/bb924375.aspx

[23] I. Molyneaux, *The Art of Application Performance Testing*. Sebastopol, CA: O'Reilly Media, Inc., 2015, 255 p.

[24] K. Pohl, G. Böckle, and F. J. v. d. Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2005, 467 p.

[25] Research Triangle Institute, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep. Planning Report 02-3, May 2002. Saatavissa: http://www.nist.gov/director/planning/upload/report02-3.pdf

[26] O. Taipale, J. Kasurinen, K. Karhu, and K. Smolander, "Trade-off between automated and manual software testing," *International Journal of System Assurance Engineering and Management*, vol. 2, 2011.

[27] T. Weiss. We Analyzed 30,000 GitHub Projects - Here Are The Top 100 Libraries in Java, JS and Ruby. Viitattu 28.02.2016. Saatavissa: http://blog.takipi.com/we-analyzed-30000-github-projects-here-are-the-top-100-libraries-in-java-js-and-ruby/

LIITE 1. ESIMERKKITOTEUTUS TESTIASKELEESTA

```
1  public class RemoteCallStep extends BaseTestStep {
2
3    private TestRemoteCallService service;
4
5    @Before
6    public void setUp() {
7      service = getService(TestRemoteCallService.class);
8    }
9
10   @Test
11   public void remoteMethodShouldBeCalledRepeatedly() {
12     final Map<String, Object> result = new TreeMap<>();
13     int callCounter = 0;
14     final int testDuration = getParameter("test.duration", int.class);
15     final long endTime = getEndTime(testDuration);
16     while (getCurrentTime() <= endTime) {
17       result.put("remote.call.delay." + callCounter, measureRemoteCallTime());
18       callCounter++;
19     }
20     result.put("remote.call.delays.count", callCounter);
21     writeResult(result);
22   }
23
24   private long measureRemoteCallTime() {
25     startStopWatch();
26     // Test method returns true as a sign of successful remote call.
27     // In case the call fails, JUnit considers this test to be a failure
28     // because of the assertion failure.
29     final boolean success = service.testMethod();
30     final long delay = stopStopWatch();
31     assertTrue(success);
32     return delay;
33   }
34 }
```

LIITE 2. ESIMERKKITOTEUTUS TESTITAPAUKSESTA

```
1  @MXBean
2  public interface RemoteCallCase {
3    // The test case name shown in the test report.
4    String TEST_CASE_NAME = "Remote call test";
5
6    // Test step result key for the number of measured remote call delays.
7    String REMOTE_CALL_DELAYS_COUNT = "remote.call.delays.count";
8
9    // Test step result prefix for a remote call delay in milliseconds.
10   // This prefix will be appended with a zero based ordinal number suffix.
11   String REMOTE_CALL_DELAY_PREFIX = "remote.call.delay.";
12
13   // Method which will be called when test run is started.
14   String execute(int parameterAmount, int parameterSize, int testDuration);
15 }
16
17 public class RemoteCallCaseImpl extends BaseTestCase implements
       RemoteCallCase {
18
19   private static final String REMOTE_CALL_STEP = "com.example.RemoteCallStep";
20
21   @Override
22   public String execute(final int parameterAmount, final int parameterSize,
         final int testDuration) {
23     return runAsynchronously(new Runnable() {
24
25       @Override
26       public void run() {
27         final Map<String, Integer> agentCounts = getAgentCounts();
28         final Map<String, Object> testStepParameters = new TreeMap<>();
29         testStepParameters.put("parameter.count", parameterAmount);
30         testStepParameters.put("parameter.size", parameterSize);
31         testStepParameters.put("test.duration", testDuration);
32         triggerTestStepExecution(REMOTE_CALL_STEP, testStepParameters);
33         sleep(getIntParameter("short.safe.delay") + 1000 * testDuration +
               getIntParameter("max.iteration.delay"));
34         triggerTestStepsResultsReporting(REMOTE_CALL_STEP);
35         sleep(getIntParameter("max.report.delay"));
36         analyze(agentCounts, parameterAmount, parameterSize, testDuration);
```

```
37        }
38      });
39    }
40
41    private void analyze(Map<String, Integer> agentCounts, int parameterAmount,
          int parameterSize, int testDuration) {
42      final List<Parameter> parameters = getParameters(parameterAmount,
            parameterSize, testDuration);
43      final Map<String, Map<String, Object>> methodCallResults =
            getTestStepResults(REMOTE_CALL_STEP);
44      final List<Failure> failures = Failure.fromResults(REMOTE_CALL_STEP,
            methodCallResults);
45      final List<Measurement> measurements = getMeasurements(methodCallResults);
46      final Builder builder = new Builder(this, TEST_CASE_NAME);
47      builder.agents(agentCounts);
48      builder.parameters(parameters);
49      builder.failures(failures);
50      builder.measurements(measurements);
51      setSummary(TEST_CASE_NAME, builder.build());
52    }
53
54    private List<Parameter> getParameters(int parameterAmount, int
          parameterSize, int testDuration) {
55      final List<Parameter> parameters = new LinkedList<>();
56      parameters.add(new Parameter("Parameter count", parameterAmount));
57      parameters.add(new Parameter("Parameter size (byte)", parameterSize));
58      parameters.add(new Parameter("Test duration (sec)", testDuration));
59      return parameters;
60    }
61
62    private List<Measurement> getMeasurements(Map<String, Map<String, Object>>
          methodCallResults) {
63      final List<Measurement> measurements = new LinkedList<>();
64      final Measurement remoteMeasurement = new Measurement("All remote call
            processing delays (ms)");
65      for (Entry<String, Map<String, Object>> agentResultEntry :
            methodCallResults.entrySet()) {
66        final String agent = agentResultEntry.getKey();
67        final Map<String, Object> agentResult = agentResultEntry.getValue();
68        final Measurement agentMeasurement = new Measurement("Remote call
              processing delay of " + agent + " (ms)");
69        final Integer callCount = (Integer)
              agentResult.get(REMOTE_CALL_DELAYS_COUNT);
70        if (callCount != null) {
```

```
71          for (int i = 0; i < callCount; i++) {
72            final Long callDelay = (Long)
                  agentResult.get(REMOTE_CALL_DELAY_PREFIX + i);
73            if (callDelay != null) {
74              agentMeasurement.addValue(callDelay);
75              remoteMeasurement.addValue(callDelay);
76            }
77        }
78        measurements.add(agentMeasurement);
79      }
80      measurements.add(remoteMeasurement);
81      return measurements;
82    }
```